The purpose of this deliverable is to describe the Y1 progress of WP3 of DIACHRON. The deliverable reports on the theoretical background upon which the development of the WP3 services will be based, namely Change Detection, Change Monitoring and Propagation, Repairing and Cleaning. In particular, we define the related problems, explain the associated challenges and describe our approach to solving them.
<table>
<thead>
<tr>
<th>Ver.</th>
<th>Date</th>
<th>Contributor(s)</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.1</td>
<td>January 30th, 2014</td>
<td>Giorgos Flouris</td>
<td>Table of contents, initial assignments</td>
</tr>
<tr>
<td>0.2</td>
<td>February 4th, 2014</td>
<td>Michalis Chortis</td>
<td>Repairing (preliminaries, diagnosis algorithm details)</td>
</tr>
<tr>
<td>0.3</td>
<td>February 14th, 2014</td>
<td>Ioannis Chrysakis</td>
<td>Representing Changes (ontology of changes)</td>
</tr>
<tr>
<td>0.4</td>
<td>February 28th, 2014</td>
<td>Giorgos Flouris, Ioannis Chrysakis, Yannis Roussakis, Michalis Chortis</td>
<td>First draft of Introduction, Change Detection and Representation, and Repairing</td>
</tr>
<tr>
<td>0.5</td>
<td>March 7th, 2014</td>
<td>All involved</td>
<td>First complete draft (improvements pending)</td>
</tr>
<tr>
<td>0.8</td>
<td>March 21st, 2014</td>
<td>All involved</td>
<td>Second complete draft</td>
</tr>
<tr>
<td>0.9</td>
<td>March 24th, 2014</td>
<td>All involved</td>
<td>Ready for the internal review process</td>
</tr>
<tr>
<td>1.0</td>
<td>March 30th, 2014</td>
<td>All involved</td>
<td>Final version</td>
</tr>
</tbody>
</table>
# Table of Contents

1 INTRODUCTION .......................................................... 6  
1.1 A DYNAMIC DATA WEB .............................................. 6  
1.2 THE DIACHRON CHALLENGE ......................................... 6  
1.3 OBJECTIVES OF THIS DELIVERABLE ............................. 6  

2 CHANGE DEFINITION, REPRESENTATION AND DETECTION ................. 9  
2.1 DEFINING CHANGES ................................................ 9  
\hspace{1em}2.1.1 BACKGROUND ........................................... 9  
\hspace{1em}2.1.2 CHANGE CHARACTERISTICS IN THE CONTEXT OF DIACHRON ........ 10  
\hspace{1em}2.1.3 BASIC CHANGES ......................................... 12  
\hspace{1em}2.1.4 SIMPLE CHANGES ......................................... 12  
\hspace{1em}2.1.5 COMPLEX CHANGES ....................................... 15  
\hspace{1em}2.2 REPRESENTING CHANGES ..................................... 16  
\hspace{1em}2.2.1 MOTIVATION FOR THE ONTOLOGY OF CHANGES .............. 16  
\hspace{1em}2.2.2 INTRODUCTION TO THE ONTOLOGY OF CHANGES ............ 17  
\hspace{1em}2.2.3 LEVEL CONCEPTS ......................................... 17  
\hspace{1em}2.2.4 BASIC CHANGES MODELLING ................................ 18  
\hspace{1em}2.2.5 SIMPLE CHANGES MODELLING ............................. 20  
\hspace{1em}2.2.6 COMPLEX CHANGES MODELLING ........................... 21  
\hspace{1em}2.3 DETECTING CHANGES .......................................... 24  
\hspace{1em}2.3.1 CHANGE DETECTION MODULE ............................... 24  
\hspace{1em}2.3.2 LOW-LEVEL CHANGES DETECTION .......................... 25  
\hspace{1em}2.3.3 BASIC CHANGES DETECTION ................................ 26  
\hspace{1em}2.3.4 SIMPLE CHANGES DETECTION ............................. 27  
\hspace{1em}2.3.5 COMPLEX CHANGES DETECTION ........................... 29  

3 CHANGE MONITORING AND PROPAGATION ............................... 31  
3.1 INTRODUCTION .................................................. 31  
3.2 RELATED WORK .................................................. 31  
3.3 REQUIREMENTS .................................................. 33  
\hspace{1em}3.3.1 CREATE A MONITORING TASK ............................... 33  
\hspace{1em}3.3.2 MODIFY A MONITORING TASK .............................. 33  
\hspace{1em}3.3.3 DELETE A MONITORING TASK .............................. 34  
\hspace{1em}3.3.4 START A MONITORING TASK ............................... 34  
\hspace{1em}3.3.5 STOP A MONITORING TASK ................................ 34  
\hspace{1em}3.3.6 PAUSE A MONITORING TASK ............................... 34  
\hspace{1em}3.3.7 REQUEST MONITORING TASK CONFIGURATION ............... 34  
\hspace{1em}3.3.8 LIST MONITORING TASKS .................................. 34  
\hspace{1em}3.3.9 SUBSCRIBE TO A MONITORING TASK ....................... 34  
\hspace{1em}3.3.10 UNSUBSCRIBE FROM A MONITORING TASK ............... 35  
\hspace{1em}3.3.11 PROPAGATE NOTIFICATIONS ............................... 35  
3.4 ARCHITECTURE ................................................ 35  
3.5 WORKFLOWS ................................................ 36  

4 REPAIRING AND CLEANING .......................................... 42  
4.1 REPAIRING ................................................ 42  
\hspace{1em}4.1.1 INTRODUCTION ........................................... 42  
\hspace{1em}4.1.2 RELATED WORK ON DIAGNOSIS AND REPAIR ............... 43  
\hspace{1em}4.1.3 PRELIMINARIES ........................................... 44  
\hspace{1em}4.1.4 OVERVIEW OF THE VALIDATION AND REPAIRING SERVICE ARCHITECTURE .......................... 46
Grant Agreement No. 601043

4.1.5 Diagnosis Component Details ........................................... 47
4.1.6 Repair Component Details .............................................. 50
4.1.7 Repairing in DIACHRON .................................................. 51
4.2 Cleaning ............................................................................. 53
  4.2.1 Motivation ..................................................................... 53
  4.2.2 Classification of Cleaning Problems ................................. 54
  4.2.3 Cleaning Module Architecture ........................................ 55

5 Conclusion ............................................................................ 59

A Basic Changes in the DIACHRON Language of Changes ............... 63

B Proposed Simple Changes for Pilots ......................................... 67
  B.1 Relational Model .............................................................. 67
  B.2 Multidimensional Model .................................................... 71
  B.3 RDF(S) Model ................................................................. 76
    B.3.1 Simple Changes Proposed by the Scientific Pilot ............. 76
    B.3.2 Other Proposed Simple Changes ................................... 78

C Constraints Supported by DL-Lite$_A$ ..................................... 88

D Data Constraints by the Pilots .................................................. 89
TABLE OF FIGURES

1 Partitioning of Low-level Changes into High-level Ones ........................................ 11
2 Simple Changes Selection ......................................................................................... 15
3 Parameter Filters Definitions ..................................................................................... 16
4 Parameter Filters Definitions ..................................................................................... 16
5 Top-level Concepts .................................................................................................... 18
6 Basic Changes Definition .......................................................................................... 19
7 Basic Changes Detection ........................................................................................... 19
8 Simple Changes Definition ......................................................................................... 20
9 Simple Changes Detection .......................................................................................... 21
10 Complex Changes Definition ...................................................................................... 22
11 Complex Change Detection ......................................................................................... 24
12 Space, Time and Synchronization Decoupling [19] ...................................................... 32
13 Monitoring and Propagation Service Architecture ..................................................... 36
14 Workflow Diagram for Creating a Monitoring Task .................................................... 37
15 Workflow Diagram for Modifying a Monitoring Task ................................................... 37
16 Workflow Diagram for Deleting a Monitoring Task .................................................... 38
17 Workflow Diagram for Starting a Monitoring Task .................................................... 39
18 Workflow Diagram for Stopping a Monitoring Task ................................................... 39
19 Workflow Diagram for Subscribing to a Monitoring Task .......................................... 40
20 Workflow Diagram for Propagating Notifications, when the Monitoring Task Runs on Demand ................................................................. 40
21 Workflow Diagram for Propagating Notifications, when the Monitoring Task Runs Periodically ................................................................. 41
22 The DL-Lite Family and Relations .............................................................................. 44
23 Diagnosis and Repair Module Architecture ............................................................... 46
24 Example of an interdependency graph ...................................................................... 50
25 Architecture of DIACHRON Cleaning Module ........................................................... 56
26 Structure of the Quality Report .................................................................................. 57

LIST OF TABLES

1 Pilots and Data Models ................................................................................................. 13
2 High Level Information on the Simple Changes .......................................................... 13
3 Changes in Pilot’s Model ............................................................................................ 14
4 Changes in DIACHRON’s Model .................................................................................. 14
5 Examples of Errors Related to the Constrains on Literals ............................................ 55
6 Quality Problems ........................................................................................................ 58
7 Proposed Simple Changes for the Relational Model (Entity: Table) .............................. 68
8 Proposed Simple Changes for the Relational Model (Entity: Column) ......................... 69
9 Proposed Simple Changes for the Relational Model (Entity: Tuple) ............................ 70
10 Proposed Simple Changes for the Multidimensional Model (Entity: Measure) ............. 71
11 Proposed Simple Changes for the Multidimensional Model (Entity: Dimension) ......... 72
12 Proposed Simple Changes for the Multidimensional Model (Entity: Fact Table) ........ 72
13 Proposed Simple Changes for the Multidimensional Model (Entity: Codelist/Hierarchy) .............................................................................................................. 73
14 Proposed Simple Changes for the Multidimensional Model (Entity: Attribute) .......... 74
15 Proposed Simple Changes for the Multidimensional Model (Entity: Observation) ....... 75
16 High Level Information on the Simple Changes of the Scientific Pilot ......................... 76
17 Simple Changes in the Model of the Scientific Pilot .................................................... 76
18 The Scientific Pilot’s Simple Changes in DIACHRON’s Model .................................. 77
19 Constraints Supported by DL-LiteA .......................................................................... 88
20 Constraints for the Scientific Pilot .............................................................................. 90
21 Constraints Appearing in the Current Version of EFO . . . . . . . . . . . . . . . . . . . . . . . . } 91
1 Introduction

1.1 A Dynamic Data Web

The unconstrained publication of data on the Data Web has given rise to the Linked Data\(^1\) initiative, which encourages the publication and interlinking of datasets in the Semantic Web\([9, 5, 24]\). The amount of publicly available data in the Data Web constantly increases; currently, more than 2000 datasets are online, comprising a knowledge space which consists of more than 61 billion RDF triples\(^2\). Such datasets include ontologies created from Wikipedia\(^3\) or other sources\(^4\), data from e-science\(^5\), most notably in the area of life sciences\(^6\), Web 2.0 information mashups\([27]\), and others. The main value of the data published in the Data Web stems from the unmoderated nature of data publication, interlinking and reuse. This increases the added-value of interlinked datasets by identifying unknown correlations and relationships, and by allowing the re-use of concepts and properties.

Dynamism is an indispensable part of the Data Web, because datasets are constantly evolving for a number of reasons, such as the inclusion of new experimental evidence or observations, or the correction of erroneous conceptualizations\([43]\). The open and chaotic nature of the Web makes it impossible to keep track of who uses (i.e., links to) a given dataset, or what are the effects of a given change to interrelated datasets; this is in contrast to closed settings, where every change in a dataset is automatically propagated to all related parties. In fact, most datasets in the Data Web evolve without any notification whatsoever, and even timestamps are generally absent in this context\([40]\). Therefore, the only way for a remote observer to identify the evolution of a dataset is through crawling and comparison with the last known version.

1.2 The DIACHRON Challenge

In this setting, the value of many interlinked datasets degrades, as they may refer to obsolete versions of other datasets, and/or not fully exploit the knowledge in the new version. In addition, access to previous versions should be allowed to guarantee that all related applications will be able to seamlessly continue operations and upgrade to the new version at their own pace, if at all. Therefore, the evolution of datasets poses several research problems, which are related to the identification, computation, storage, crawling, annotation, cleaning, repairing, assessment and management of the evolving versions and the changes that led to/from them; addressing these problems is the main research aim of DIACHRON.

In particular, the approach of DIACHRON is based on the realization that in order to maintain the data value of linked datasets, one should keep them constantly accessible and integrated into a larger framework, where the changes are treated as first-class citizens. This gives rise to the notion of *diachronic data*, which refers to treating each dataset as one single, evolving entity, rather than as several static versions of one entity. To achieve this, changes should be raised at the same level as data, to allow seamless annotating, querying, management and understanding of the evolution process.

WP3 of DIACHRON is critical in this larger picture, because it deals with the fundamental problems of identifying, recording, monitoring and propagating the changes, as well as with restoring certain aspects of data quality (namely, validity, consistency and correctness), which are often jeopardized due to the open evolution of remote datasets. These problems will be addressed through the following DIACHRON services: change detection, change monitoring and propagation, repairing and cleaning.

1.3 Objectives of this Deliverable

The Y1 progress of WP3 is described in this deliverable, which contains the theoretical background upon which the development of the WP3 services will be based. This is a report deliverable, so our focus is on defining the

---

\(^1\)http://linkeddata.org/
\(^2\)http://stats.lod2.eu/
\(^3\)http://www.dbpedia.org
\(^4\)http://www.informatik.uni-trier.de/~ley/db
\(^5\)https://www.ebi.ac.uk/services/all
\(^6\)http://www.geneontology.org/
problems and proposing appropriate solutions; details on the implementation of said services will appear in the
prototype deliverable D3.2, due on M16 of the project. In the following, we list the problems addressed, define
them briefly and point to the appropriate sections in the text where the reader can find more details.

**Change Definition.** A fundamental problem that is related to understanding the evolution of datasets, is the
identification and definition of the set of changes that should be used for describing the evolution of the dataset as a
whole, or of the entities that comprise it (called the *language of changes*). The defined changes should correspond
to the “evolution primitives” of the dataset/entity under consideration. In our proposal, a change is defined through
a series of attributes, such as its parameters and the associated conditions for detection; defining a change amounts
to providing specific values to these attributes. In addition, changes are organized in three major types, namely,
*basic*, *simple* and *complex*, each of which represents changes with a certain granularity and role in the model. More
details on the above are provided in Section 2.1. This work corresponds to part of the work associated with T3.1
of the DoW.

**Change Representation.** The problem of change representation is dealing with providing the appropriate struc
tures and models for storing the changes in an appropriate format. In the context of DIACHRON, this problem
becomes much more involving than usual, because the changes are first-class citizens and should be stored at the
same archive (and using the same technologies) as the actual data. In addition, we need to be able to support
queries that are related not only to the data itself, but are aiming to understand the dynamics and evolution of
datasets or entities. Such queries would involve both the data and the changes (e.g., “return all countries for which
the unemployment rate of their capital city increased at a rate higher than the average increase of the country as
a whole, between versions V1 and V2”), and cannot be handled in archives that separate the data itself from the
information on changes. To achieve these objectives, the change representation model is based on an *ontology of
c changes* represented in RDF, in a way that is compatible with the DIACHRON model described in Deliverable
D1.3 [35]. More details on our proposal for representing changes can be found in Section 2.1. This work is
associated with Task 3.1 of the DoW.

**Change Detection.** The problem of change detection refers to the problem of identifying the changes that
occurred between any two given versions (and led from one version to the other). Said detection occurs *after* the
change has happened, i.e., the system assumes no knowledge on the change process itself, and the only input for
performing the detection is the content of the two versions (of the dataset or entity) to be compared. The returned
changes should be changes from the defined language of changes. More details on the change detection process
appear in Section 2.3. Change detection is part of the work associated with Task 3.1 of the DoW.

**Change Monitoring and Propagation.** The *monitoring and propagation service* provides publish-subscribe
functionality for creating dataset monitoring tasks. In particular, the user registers the dataset of her interest and
the desired time period that the dataset should be probed for changes, and the monitoring service monitors these
datasets for change, by periodically retrieving the datasets and detecting (using the change detection service)
whether any changes have happened; if so, then the change propagation service is used to propagate these changes
to the subscribed users. More details on change monitoring and propagation appear in Section 3. This work is
associated with Task 3.2 of the DoW.

**Repairing.** *Repairing* deals with the problem of identifying and resolving invalidities in datasets. An *invalidity*
is defined as a violation of a certain constraint associated with the underlying data, such as the requirement for
two concepts to be disjoint. The repairing service of DIACHRON will consider constraints of logical nature, in
particular constraints that can be expressed in the language *DL-LiteA* [12]. It will provide an efficient methodology
for identifying invalidities (taking into account *DL-LiteA* reasoning), as well as for resolving them in a manner that
has the least impact (in terms of lost knowledge) from the dataset. More details on repairing appear in Section 4.1.
This work is associated with Task 3.3 of the DoW.
Cleaning. Cleaning is the process of identifying anomalies in the data and then replacing, modifying, or deleting them in order to improve the quality of data. Anomalies include incomplete, incorrect, inaccurate, irrelevant literals or facts that reduce the number of properties known about an entity, or that lead to a wrong representation of an entity. Cleaning is a challenging task, which cannot be performed fully automatically, because detecting and correcting inconsistencies in data often requires detailed domain knowledge. In Section 4.2, we first give an overview of the quality problems that cleaning indents to solve and then describe the three steps of the cleaning process, namely detecting problems, automatic cleaning and semi-automatic cleaning. This work is associated with Task 3.3 of the DoW.
2 Change Definition, Representation and Detection

In this section, we define the framework upon which our definition of the changes in DIACHRON will be based. In particular, we will first describe how a change should be defined (i.e., the specifications of the semantics of the changes) and give the types of supported changes; then we will explain how these changes will be represented in a way that will allow their seamless integration with the actual data (which is necessary in order to support the requirements of the DIACHRON use cases); and finally we will describe how changes will be detected via the change detection service of DIACHRON.

2.1 Defining Changes

In order to provide the desired functionality regarding changes and evolution in DIACHRON, we need first of all to provide a formal specification/definition of what a change is.

Further, we need to study what types of changes are necessary in the context of DIACHRON and define a language of changes, which is essentially a set of changes that the entire DIACHRON system understands and supports.

In this section, we will deal with the above two problems. Our approach is based on previous work by FORTH, which was presented in [34]. We briefly revisit the definitions there and then explain how we deviate from that work for the purposes of DIACHRON.

2.1.1 Background

In [34], the purpose was to provide a change detection tool, which, given two (subsequent) dataset versions \( V_1, V_2 \), would produce their delta, i.e., a formal description of the changes that were made to get from \( V_1 \) to \( V_2 \). A delta is based on a language of changes, i.e., a set of formal definitions, under some common template, of the changes that the delta could contain (and, subsequently, that the change detection tool understands and detects).

In that context, the focus lied on RDF datasets, so in its simplest form, a language of changes consists only of changes of the form \( \text{Add}(t) \) or \( \text{Del}(t) \), for some triple \( t \); thus, a delta essentially describes the triples that were added and deleted. Such a delta is called a low-level delta [47] and defined as \( \Delta(V_1, V_2) = (V_2 \setminus V_1, V_1 \setminus V_2) \). For brevity, we use \( \Delta^+ \) for the set \( V_2 \setminus V_1 \) (the triples added to \( V_1 \) to get to \( V_2 \)) and \( \Delta^- \) for the set \( V_1 \setminus V_2 \) (the triples deleted from \( V_1 \) to get to \( V_2 \)).

Low-level changes are easy to define and have several nice properties; however, the representation of changes at the level of (added/deleted) triples, leads to a syntactic delta, which does not capture the semantics of a change and generates results that are not intuitive enough for the human user.

For example, in the RDF context, the plain deletion of an individual (class instance) would correspond to a multitude of triple deletions (namely all the triples that contain this URI, such as property instances, typing triples etc); this makes it hard for a human observer to decipher the actual changes that took place at the intuitive level. Other problems with low-level deltas are that they don’t provide useful feedback to the user; for example, if a domain of a property changed, it would be useful for the user to know whether the new domain is a subclass or superclass of the old. More examples of shortcomings related to low-level changes appear in [34].

The notion of high-level deltas was proposed to fill this gap. High-level deltas aim to describe changes at a more intuitive level, in order to make them more human-understandable. The main idea behind achieving this is to “group” low-level changes into high-level ones; thus, a high-level change corresponds to a set of low-level ones, under some conditions. Unlike previous works on high-level deltas, the approach of [34] did not forego formal rigour to achieve this aim. This is done by associating high-level changes with a formal definition that specifies their exact semantics.

More specifically, a change \( c \) in [34] is defined as a quadruple \( < \delta^+, \delta^-, \mathcal{M}, \phi > \) where:

- \( \delta^+ \) is a set containing the low-level changes that are expected to be found in \( \Delta^+ \) in order for \( c \) to be detectable.
- \( \delta^- \) is a set containing the low-level changes that are expected to be found in \( \Delta^- \) in order for \( c \) to be detectable.
- \( \mathcal{M} \) is a set of mappings that should exist between URIs in \( V_1 \) and \( V_2 \) in order for \( c \) to be detectable.
- \( \phi \) is a condition (logical formula) related to \( V_1 \) and/or \( V_2 \) that is required to be true in order for \( c \) to be detectable.

Some explanations on this structure are in order. First of all, \( \delta^+, \delta^- \) are the low-level changes that the high-level one “groups” under it. Each mapping in \( \mathcal{M} \) corresponds to an association of a pair of URIs appearing in \( V_1 \) and \( V_2 \); mappings are meant to capture changes like renaming, where two different URIs (in two different datasets) correspond to the same real-world entity. Finally, \( \phi \) is a logical formula that allows one to discriminate between similar changes, e.g., in the example above with the change in the domain of a property, the condition \( \phi \) could be used to discriminate between a generic Change Domain operation, as opposed to the more specific Generalize Domain, Specialize Domain.

The above definition was used in [34] to propose a language of changes. This language consisted of three types of changes, namely basic, composite and heuristic. Basic changes are simple, fine-grained ones, composite changes are more coarse-grained structural changes, whereas heuristic changes refer to changes in terminology (e.g., renaming).

The proposed language of [34] was shown to satisfy the intuitively expected properties of completeness and unambiguity, which are illustrated in Figure 1. In a nutshell, completeness guarantees that the reported delta will not “miss” any change, whereas unambiguity guarantees that the reported delta will not report “twice” any change (see [34] for the formal definitions).

To understand better these terms, see Figure 1. Consider a low-level delta \( \Delta \) which contains the low-level changes \( t_1, t_2, \ldots, t_5 \) (either as added or deleted – this is irrelevant for this discussion). In the left box, we visualize a case where the definition of the high-level change \( HLC1 \) captures the change described by \( t_1, t_4, t_5, t_6 \) in the low-level delta, whereas the definition of \( HLC2 \) captures the change described by \( t_2, t_3 \).

In this scenario, no low-level change is left “undescribed” by the high-level delta, and all low-level changes are described once in the high-level delta. Essentially, the definition of \( HLC1 \), \( HLC2 \) guarantees that the changes in the low-level delta will be completely partitioned into high-level ones in a complete and unambiguous way, so the “grouping” of low-level changes into high-level ones (as described above) is “perfect”.

On the other hand, if we consider the scenario in the upper right part of Figure 1, we notice that, under this definition of basic changes, the low-level change \( t_6 \) is not mapped into any high-level one (incompleteness). This means that the reported high-level delta will miss one, potentially important, information, i.e., that \( t_6 \) changed.

The lower right part of Figure 1 shows a case of ambiguity, as the low-level change \( t_4 \) is associated with both high-level changes. This creates some ambiguity as to whether \( t_4 \) should contribute to the detection of \( HLC1 \) or \( HLC2 \). Thus, it is unclear whether \( HCL1 \), \( HLC2 \), or both, should be reported.

### 2.1.2 Change Characteristics in the Context of DIACHRON

Now let’s see how the above work can be applied in the context of DIACHRON. To do so, we will examine the special characteristics of DIACHRON which stem, firstly, by the fact that DIACHRON is meant to be used by different pilots that use different data models (i.e., not necessarily RDF) and, secondly, by the fact that DIACHRON services (including change detection) will not work on the original (raw) data, but on the underlying DIACHRON back-end, which causes some transformations on the raw data in order to fit with the DIACHRON model [35].

Our first conclusion related to the changes to be defined in DIACHRON, is that all the shortcomings related to low-level changes are also present in the DIACHRON context, and, in fact, even more alleviated. The reason for this is the fact that the storage of information is done in a verbose manner via the DIACHRON model [35], which is different than the original dataset format. Thus, a minor change in the raw data would correspond to several low-level changes in the underlying model.

Further, low-level changes (i.e., a triple addition or deletion) in the DIACHRON model may often be meaningless for the raw data, as the pilots will not be able to understand what it corresponds to in their original dataset; further in most cases, it may only correspond to part of a change that the pilots view as one indivisible change (e.g., the deletion of a tuple in a relational table), which would cause confusion. For these reasons, one of the main design decisions was that our language of changes will be a high-level one.

Given this decision, the properties of completeness and unambiguity become relevant. Our decision in this respect is that these properties must also hold for the DIACHRON language of changes, in order to allow a complete
and unambiguous reporting of the occurring changes.

Another requirement stems from the existence of the three pilots that must be supported by the DIACHRON platform, each of which employs a different data model. To support all three pilots, the DIACHRON language of changes should be actually understandable by the pilots, i.e., it should correspond to changes that refer to the original (raw) data, rather than the transformed data in the DIACHRON model, despite the fact that the actual detection will be performed on top of the DIACHRON model.

Thus, our proposed language of changes should contain a different set of changes for the relational, multidimensional and ontological setting, so that they make sense for each pilot. For example, for the enterprise pilot (relational data), one would expect changes like Delete_Column, Add_Tuple or Split_Table.

To avoid having to provide three different templates for the definition of changes, and given that all these data models will be eventually described using the DIACHRON model, which is represented in RDF, we decided that the semantics of changes will also be defined in RDF terms. Thus, even though the defined changes for the pilots will be different (and using the pilots’ terminology), the internal definition of the change will be done in RDF terms. This will allow the same implementation of the change detection service to be applicable for all three.

In the context of [34], and also more generally in the context of most evolution management tools, we are only interested in reporting/managing changes at the level of a dataset. This is in contrast to the DIACHRON case, where we are also interested in changes being reported between entities [35], i.e., parts of a dataset. For example, in the relational setting, one may want to identify changes at the level of a column, or table. Note that the set of changes related to the “column” entity are different than those related to the “dataset” entity, so a different set of changes should be defined per entity.
This problem is addressed in the same way as above: all changes are defined in terms of the underlying DIACHRON data model (i.e., in RDF), and each change is associated (through its definition) with the entity to which it is applicable.

An additional related requirement is to allow pilots to create (i.e., define) changes that were not foreseen at design-time. This would give an extra flexibility to the user to describe more specialized, coarse-grained changes that are relevant for a certain application, but not useful enough in the general case so as to be considered part of the main language of changes. This essentially means that the language of changes should be extensible at run-time.

To support all the above requirements, we support three different types of changes, namely basic, simple and complex. Basic changes are generic, fine-grained changes related to the DIACHRON model and applicable to all pilots; simple changes are pilot-specific changes defined at design-time; whereas complex changes are pilot-specific changes defined at run-time. In the following subsections, we describe in detail the three types of changes, explain how they are defined, and give a list of the changes per type that exist in the proposed DIACHRON language of changes.

2.1.3 Basic Changes

Basic changes form the baseline changes, i.e., the most fine-grained level of changes for DIACHRON. They are generic, pilot-agnostic changes that only make sense in the context of the DIACHRON internal representation of the datasets. As such, they are for internal use only and will not be visible to the pilots. Basic changes are actually quite close to low-level changes. This is necessary in order to be able to capture all types of changes (even fine-grained ones) in a pilot-agnostic manner.

The formal definition of basic changes follows the tradition of [34], and consists of three elements \((\delta^+, \delta^-, \phi)\), whose meaning is described below:

- The sets \(\delta^+, \delta^-\) are the low-level changes which are required to be present in the low-level delta \((\Delta^+, \Delta^-)\) respectively in order for said change to be detected.
- The formula \(\phi\) is a logical condition which must be satisfied in order for the change to be detected.

The list of basic changes that will be included in the DIACHRON language of changes, as well as their formal semantics and definitions, have already been decided and appear in Appendix A. The considered changes are actually taken from [34], and they correspond to the basic changes defined in that paper. The reason for this choice is that said changes have been proved to satisfy completeness and unambiguity [34], which guarantees that any change in any of the pilots will be describable in a perfect manner using basic changes alone.

As we will explain in the section describing the simple changes below, this property is not guaranteed to hold for the simple changes of any of the pilots; thus, the use of basic changes is necessary to guarantee the fundamental required properties for the language of changes. Other than that, basic changes are not useful, and will be dropped in the future should simple changes allow a perfect description of the changes performed in the pilots’ data models.

2.1.4 Simple Changes

In contrast to basic changes which refer to DIACHRON’s internal data model, simple changes are pilot-specific, and their purpose is to describe the changes which are of interest to the pilots for their domain. Simple changes must be defined at design time, and represent evolution primitives that are relevant for the data model of the corresponding pilot. Table 1 lists the data pilots in DIACHRON project and their corresponding use cases and data models.

Because of the fact that each pilot works on a different data model, we have designed some uniform guidelines and templates which must be followed to define properly the simple changes. These guidelines will be used as a general-purpose methodology to determine the useful simple changes of each pilot and their precise semantics. In the following tables (namely Tables 2, 3 and 4), we show, by means of two examples, how this methodology is used to disambiguate the semantics of the changes in order to be properly formalized. In particular, these tables explain what the changes correspond to at the intuitive level, but also at the level of the pilot’s data model, and, most importantly, at the level of the DIACHRON internal model. The latter will be used to identify the process
Table 1: Pilots and Data Models

<table>
<thead>
<tr>
<th>Use Case</th>
<th>Partner</th>
<th>Data Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open Data</td>
<td>DATAPUB, DM</td>
<td>Multi-dimensional</td>
</tr>
<tr>
<td>Enterprise</td>
<td>BROX</td>
<td>Relational</td>
</tr>
<tr>
<td>Scientific</td>
<td>EMBL</td>
<td>Ontological (RDF)</td>
</tr>
</tbody>
</table>

through which the change will be actually detected by the change detection service (see Section 2.3). The changes we will consider are Add new label and Mark as obsolete, which refer to the biological data from EMBL.

Table 2 is used to identify some generic information on said changes and shows the relevant information for the simple changes Add new label and Mark as obsolete. It has the following fields:

- The field **Change with parameters** shows the change’s name and parameters.
- The field **Priority** determines the priority of changes, and is quite important for detection purposes. At the moment, we have no guarantees that the actual simple changes defined in each model will possess the property of unambiguity. To address this problem, we plan to assign priorities in simple changes, which would solve ambiguity issues when two or more simple changes could potentially be associated with the same low-level change (cf. Figure 1). Using priorities, one could decide where to associate the “shared” low-level change (and, consequently, which simple change to report as a detectable change). Note that priority is not an indicator of importance, just a method to decide which change should be detected when there are ambiguities.
- The field **Parameters (in details)** describes the parameters of the change in more detail; note that the parameters are related to the underlying model (e.g., for RDF we could have classes, properties etc).
- The field **Intuition** describes informally the effect of the change in the dataset.

Table 2: High Level Information on the Simple Changes

<table>
<thead>
<tr>
<th>Change with parameters</th>
<th>Priority</th>
<th>Parameters (in details)</th>
<th>Intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add new label(A, &quot;label&quot;)</td>
<td>3</td>
<td>A: the class which obtained the label label: the new label</td>
<td>This corresponds to the addition of a triple associating A with a new label</td>
</tr>
<tr>
<td>Mark as obsolete(A)</td>
<td>1</td>
<td>A: the class which is rendered obsolete</td>
<td>This corresponds to the subsumption of A with the class Obsolete</td>
</tr>
</tbody>
</table>

The next table (Table 3), determines the effects of the simple change in the pilots’ data model for our example changes. The used fields are explained below:

- The field **Effects in pilot’s model**, describes the change’s impact on the new version of the dataset in terms of added and deleted information from the pilot’s data model (in this case, triples, as the model of EMBL is RDF).
- The field **Presence** determines whether the changes of the previous field are mandatory or optional. This is needed because, in the case of the simple changes of the DIACHRON pilots, the need for mandatory and optional changes emerged, which did not appear in previous language of changes (e.g., [34]). In a nutshell, a mandatory change is one which must be found in the low-level delta in order for the simple change to be...
detected; an optional change need not appear in the low-level delta for the simple change to be detected, but, if it appears (and the simple change is detected), it will be associated with said simple change. The field Presence of Table 3 encodes this information, i.e., whether each of the low-level changes in the column Effects in pilot’s model is mandatory or optional.

- The field Condition describes the conditions that should be true in the current or previous version of the dataset in order for the change to be detected. Even though no conditions are necessary for the simple changes shown in this example, this need arises in other change definitions. For example, it would make sense to require that the class A in Add_new_label should be a class that exists in both versions; in this case, the addition of labels for new classes (i.e., recently added classes that don’t appear in the old version) should be described as part of the simple change that adds the class (Add,Class – cf. Appendix B). This is an example of a design decision regarding the definition of changes that needs to be taken in the near future.

Table 3: Changes in Pilot’s Model

<table>
<thead>
<tr>
<th>Change with parameters</th>
<th>Effects in pilots model</th>
<th>Presence</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_new_label(A,“label”)</td>
<td>added: (A, rdfs : label, “label”)</td>
<td>mandatory</td>
<td>N/A</td>
</tr>
<tr>
<td>Mark_as_obsolete(A)</td>
<td>added: (A, rdfs : subClassOf, Obsolete)</td>
<td>mandatory</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The above tables are only useful for the pilots and the research partners to understand the semantics of the changes and to communicate. The important part for the formal definition of the changes and their support within the DIACHRON language of changes (and Change Detection service) is Table 4, in which the changes are formally defined in terms of the DIACHRON data model. To achieve this, the information encoded in Table 3 needs to be translated in terms of the DIACHRON model, i.e., in terms of the actual triples in the DIACHRON model which represent the pilot-specific items. This is done in Table 4, which contains the following fields:

- The field Effects in DIACHRON’s model, describes the change’s impact on the new version of the dataset which mapped into DIACHRON.
- The fields Presence and Condition have the exact same meaning as in Table 3 with the difference that they now refer on DIACHRON’s triples.

Table 4: Changes in DIACHRON’s Model

<table>
<thead>
<tr>
<th>Change with parameters</th>
<th>Effects in DIACHRON’s model</th>
<th>Presence</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_new_label(A, “label”)</td>
<td>added: (R1, diachron : subject, A) (R1, diachron : hasAttribute, RAtt1) (RAtt1, diachron : predicate, rdfs : label) (RAtt1, diachron : object, “label”)</td>
<td>mandatory</td>
<td>N/A</td>
</tr>
<tr>
<td>Mark_as_obsolete(A)</td>
<td>added: (R1, diachron : subject, A) (R1, diachron : hasAttribute, RAtt1) (RAtt1, diachron : predicate, rdfs : subClassOf) (RAtt1, diachron : object, Obsolete)</td>
<td>mandatory</td>
<td>N/A</td>
</tr>
</tbody>
</table>

The above tables will be filled for all simple changes in cooperation with the pilots, in order to finalize the list of simple changes admitted by the DIACHRON language of changes. This list has not been finalized yet; for a preliminary list (not yet fully defined per the above methodology), see Appendix B.
Some discussion on the connection between basic and simple changes is in order here. Given that basic changes are practically meaningless for the pilots, whereas simple ones are meant to be the evolution primitives that will describe the evolution of the pilots’ data model, one may question the usefulness of basic changes.

In fact, basic changes are not necessarily critical for the change model we use in DIACHRON. Basic changes represent a set of changes that is guaranteed to provide completeness and unambiguity. At the time of writing this deliverable, there is no guarantee that simple changes defined by pilots will have the same property, even though this will be our objective.

If simple changes turn out to be ambiguous for some pilot, the priority property of simple changes described above will be used to amend the situation. On the other hand, incompleteness will be handled by associating those low-level changes that are not properly described by the detected simple changes, with adequate basic ones. This combination will guarantee that any change performed in the pilots’ data models will be perfectly described (in the sense of completeness and unambiguity).

2.1.5 Complex Changes

Even though simple changes are meant to capture the evolution primitives of the pilots’ change, they may not be enough to optimally describe the evolution of the pilots’ datasets. There are two reasons for that. First, a pilot may be interested to have higher-level changes in the detection report, i.e., changes that group simple changes into more high-level ones in a manner similar to high-level changes being groups of low-level ones. Second, there may be cases where a user, for some specific application, requires the report of some special combination of simple changes as one single change; this may be because this combination happens often in the user’s application context, or because it is highly interesting for the user’s purposes.

Both needs are addressed using the notion of complex changes, which are pilot-specific changes that are defined at run-time by the user in order to capture some special need. As already mentioned, the purpose of complex changes is to combine multiple simple changes in order to extract useful conclusions with respect to the evolution of their data. They represent high-level, often-occurring, interesting groups of simple changes that the data curator is interested in characterizing and considering as one unit. As complex changes are totally customized there will be no guarantees (or checks) that they will be unambiguous or complete.

As complex changes are user-defined, DIACHRON will not propose any specific list; instead, they could be defined via an appropriate user interface (UI). One possible proposal for the design of this interface is shown below.

![Figure 2: Simple Changes Selection](image)

The first step in defining a complex change is to give it a name and select the simple change(s) which comprise it (see Figure 2). In order for a complex change to be detected, all the simple changes that comprise it must be also detected. Further, the user should define the parameters of the complex change and associate them with the parameters of the simple changes that comprise it. Finally, the user will optionally be able to define filters over the parameters of said simple changes, which should be true in order for the complex change to be reported as detectable (see Figure 3). Such filters can be selection filters (e.g., the value of parameter \textit{Param}_1 is set to be equal to \textit{Value} in Figure 3a), or join filters (e.g., the value of parameter \textit{Param}_1 of simple change \textit{Simple Change}_1 must be equal to the value of parameter \textit{Param}_1 of simple change \textit{Simple Change}_2 in Figure 3b).

Moreover, a user can also modify existing complex changes that he has already defined by updating the set of the selected simple changes or the filters he has applied on the simple changes’ parameters. Finally, he can...
also delete one or more defined complex changes. These functionalities may be supported in the UI as shown in Figure 4.

An example would help clarify the above. Suppose that a user is interested in defining a complex change (say _Label_Obsolete(A, “label”)_) for the case where a new label is added to a class which became obsolete during the same evolution phase.

This can be easily expressed by defining a new complex change which combines the simple changes described before (_Add_new_label_ and _Mark_as_obsolete_). Further, the user should associate the first parameter of _Add_new_label_ (i.e., the class that got the new label) with the first parameter of _Label_Obsolete_ and the second parameter of _Add_new_label_ (i.e., the new label) with the second parameter of _Label_Obsolete_. Further, a join filter should be defined to “associate” the class (first parameter) in _Add_new_label_ and _Mark_as_obsolete_; this will guarantee the intuitively expected property that the two changes should refer to the same class in order for _Label_Obsolete_ to be detectable.

### 2.2 Representing Changes

#### 2.2.1 Motivation for the Ontology of Changes

As already mentioned, changes in DIAChRON need to be treated as first-class citizens, in order to be able to perform queries analysing the evolution of datasets. Further, we should be able to perform combined queries, in
which both the datasets and the changes should be considered to get an answer. To achieve this, the representation of the changes that are detected on the data cannot be separated from the data itself.

To understand this, let’s consider the following query: “return all countries for which the unemployment rate of their capital city increased at a rate higher than the average increase of the country as a whole, between versions V1 and V2”. This is a quite complicated query. On the one hand, it requires access to the data itself (to identify countries and their capitals). Moreover, it requires being able to identify the changes describing the actual increase in the rate of unemployment for each city and country and compare it; the latter information is found in the changes, not in the dataset itself. Therefore, to answer the above queries, the changes should be stored in a structured form and their representation should include connections with the actual entities (cities or capitals) that they refer to.

To address this need, we propose a solution where changes will be represented within the DIACHRON model, as special entities. The detected changes will be associated with their respective DIACHRON entities. Thus, one will be able to perform queries like the above.

Note that the above refers to the storage of the detected changes, i.e., the ability to store that “change C1 was detected between versions V1, V2 and it referred to the entity A”. Another requirement for the representation of changes stems from the complex ones; given that complex changes are defined at run-time, we should have a structure that would store their definition. This requirement will also be addressed in the same way, by storing the definition of the complex changes (as described in the previous subsection) within the DIACHRON model.

The semantics of both the detection and the definition of changes will be supported by an ontology that describes them. Ontologies provide the functionality of maintaining (sharing and reuse) of linked datasets and allow the flexible tracking of changes with support of query capabilities in a uniform manner. In addition, the ontological representation provides a supervisory look of the detected changes and their association with the entities they refer to in the actual datasets, as well as with the definition of the complex changes. This way, it facilitates the formulation and the answering of queries that refer to both the data and their evolution, as well as the easy retrieval of the specifications associated with a custom complex change for use by the change detection algorithm (to be described in Subsection 2.3). All these can be done using standard RDF technologies (e.g., SPARQL queries) over the DIACHRON data model. In the rest of this subsection, we describe the basic concepts associated with the proposed ontology of changes, and show how it can be used to achieve the aforementioned aims.

### 2.2.2 Introduction to the Ontology of Changes

Before going into the details of our ontology of changes, let us first explain the general idea behind the proposed representation. In a nutshell, the schema level describes the definition of the changes, whereas detected changes appear at the instance level, instantiating the corresponding classes.

More specifically, at the schema level, we introduce one class for each basic, simple or complex change that is understood and considered by the language of changes and the change detection service. In addition, the schema level contains some additional classes that represent parameters. To define complex changes, they are associated (at the schema level) with the simple changes they comprise of; similar connections exist between the parameters of the complex and their comprising simple changes. Further, the definition of a complex change at the schema level contains a detailed description (through adequate properties) of its name, filters etc, as well as to the SPARQL query required for its detection (cf. Subsection 2.3).

At the instance level we store the detected changes. In particular, each detected change between any two given versions will be represented using a different instance, associated through adequate properties with its parameters and any other information that may be relevant to said change (such as the versions and the entities it refers to). All this information is stored by creating adequate instances of the corresponding classes at the schema level (which describe the change definition and its parameters); note that this is true for all types of changes (not just for complex ones).

### 2.2.3 Top-level Concepts

In the following, we describe in more detail the ontology of changes. We start with the most high-level concepts in the hierarchy, which are shown in Figure 5. We represent each change category (basic, simple and complex) with a class and there is also a general class called Change which generalizes them. The top-level class (Change) is
linked with the class Diachron_Entity through properties old_version and new_version, representing the versions among which the change was detected. This way, any detected change (which will be an implicit instance of class Change) will inherit these properties in order to store the versions to which it refers to, allowing us to identify changes referring to any two given versions.

2.2.4 Basic Changes Modelling

For each basic change considered by our language of changes, we create one class that is a subclass of the generic class Basic_Change. The parameters of each basic change are also modelled as classes linked with the corresponding change. For example, in Figure 6 we present two basic changes namely, Add_Type_Class and Add_Superclass. Let’s take Add_Type_Class, which has one parameter, the class which was inserted in the new dataset version (cf. Appendix A); as we can see in Figure 6, this is represented by connecting Add_Type_Class to a class representing said parameter (atc_n1), and the class atc_n1 with another class representing the values of this parameter (in this case, rdfs: class, because the values that this parameter takes are classes, i.e., instances of rdfs: class). The properties used for this connection are atc_p1 and atc_v1 respectively (see Figure 6).

We should mention here that we follow a naming scheme for parameters and property names; this scheme uses abbreviated information created based on the name of the change and encapsulates parameter order if the latter exists. This convention is followed for all changes and leads to unique names. A similar convention will be used for instances.

Next, in Figure 7 we show how a detected basic change is stored at the instance level. In particular, this figure shows a detected basic change of type Add_Superclass; the actual change is Add_Superclass(a1,b1), indicating that the triple (a1, rdfs: subClassOf, b1) was inserted in the new dataset version (cf. Appendix A for the definition of Add_Superclass). This change is represented by adding an instance of class Add_Superclass (named asc1) along with the triples (asc1, asc_p1, asc_n1), (asc1, asc_p2, asc_n2) to denote the parameter names of the change. Next, we insert the triples (asc_n1, asc_v1, a1), (asc_n2, asc_v2, b1) which determine the parameter values of the detected change. In addition, all the appropriate instantiations (e.g., (asc1, rdf: type, Add_Superclass)) are added, as shown in the figure. The triples which denote the entity versions which were compared will have the form:
Figure 6: Basic Changes Definition

Figure 7: Basic Changes Detection

\((asc_1, old\_version, V_1), (asc_1, new\_version, V_2)\). For clarity in the figure, these are not shown in Figure 7 (these will be omitted from the other figures in this subsection as well).
2.2.5 Simple Changes Modelling

Simple changes are defined in a similar manner to basic ones. In particular, we define one subclass of the generic class SimpleChange for each simple change supported by the change detection service (Figure 8a), and define adequate classes and properties to represent its parameters, as with basic changes. For simple changes we need
to keep some additional information about the exact type of DIACHRON entity for which this change is applicable (e.g., DIACHRON record, resource etc); this is done using a property applied_on whose range is the class Diachron_Entity. Note that this need does not arise in basic changes, which are generic. A further requirement for simple changes is the recording of their priority as discussed in Subsection 2.1.4. This feature is modelled by connecting each simple change with the data type property priority. Both properties are shown in Figure 8a. Further, Figure 8b shows the representation of the definition of two simple changes defined in the previous subsection, namely Mark_as_obsolete and Add_new_label.

![Figure 9: Simple Changes Detection](image)

Similarly to basic changes, the detection of simple changes is implemented by inserting the detected changes as instances of the corresponding simple change class. Figure 9 depicts an example of a change detection for the change Add_new_label in which the class a1 took as label the string “New Label”. This change is represented by adding an instance of class Add_new_label (named anl1) along with the triples (anl1, anl_p1, anl_v1), (anl1, anl_p2, asc_n2_1) to denote the parameter names of the change. In addition, the following triples are inserted: (anl_v1, anl_v2, “NewLabel”) which denote the parameter values of the detected change, as well as all the necessary instantiations.

2.2.6 Complex Changes Modelling

The definition of complex changes at the schema level requires a more precise modelling compared to the simple and basic ones. This is because complex changes are dynamic user-defined changes, so the full information related to the change should be available in the ontology for the change detection service to be able to identify and use; this is in contrast to the case of the basic and simple changes, where this information is known at design time and embedded in the code, so we only need to model part of the information (e.g., parameters).

Similar to the other change categories, each complex change is a subclass of the generic class Complex_Change, as shown in Figure 10. As complex changes comprise of simple changes, we use a property called comprise_of_mand to associate a complex change with the simple changes it comprises of; note that all simple changes comprising a complex one are mandatory, unlike simple changes which may comprise of low-level changes in a manda-
(a) General Case

(b) The Definition of a Sample Complex Change

Figure 10: Complex Changes Definition

tory or optional manner. Figure 10a shows the definition for the complex change \textit{ComplexChange}1 which comprises of simple change \textit{Simple\_Change}1, whereas Figure 10b shows the more specific example of the definition of \textit{Label\_Obsolete} which was discussed in Subsection 2.1.5, and which comprises of \textit{Add\_new\_label} and \textit{Mark\_as\_obsolete}.

Apart from the relation between the complex change and its simple changes, we also relate the parameters
of the complex change with the corresponding parameters of simple changes using the \textit{refersTo} property. As an example, in Figure 10b the parameters of the complex change, \textit{lo}_1, \textit{lo}_2, \textit{lo}_3 are related via property \textit{refersTo} with the parameters of its related simple changes \textit{anl}_1, \textit{anl}_2, \textit{mo}_1 respectively.

The type of values that the parameter can take can be implicitly deduced through the \textit{refersTo} property (as they must be the same with the values of the associated parameter of the simple change); nevertheless, for efficiency, we put an additional property associating each parameter with the type of values it can take (these properties are not shown in Figure 10a for elegance, but can be seen in Figure 10b).

Additionally, for each parameter of the complex change we store some extra information, namely any associated filters (via property \textit{filter}) and its visibility (via property \textit{visible}). The \textit{filter} property is necessary to record any filters (selection or join) that are related to said parameter. The visibility feature is necessary when we want to hide parameters values from the end user. For example, consider the case of \textit{Label\_Obsolete}, where the first parameter (the class that was added a new label) and the third parameter (the class that became obsolete) are required to be the same class (cf. the definition of \textit{Label\_Obsolete} in Subsection 2.1.5), as also recorded using a filter in Figure 10b; in this case it is more elegant to hide the third parameter than repeating the same information.

Another property associated with each complex change is the \textit{sparql} property. This is very important and used to record the SPARQL query that will be used to detect said complex change (cf. Section 2.3 below). The SPARQL can be generated at run-time from the information already in the ontology (the simple changes that comprise the complex one, the information on the parameters, filters, and the visibility flag); nevertheless, for efficiency, we create this query at the time when this complex change is created and ingested in the ontology, in order for the change detection service to have it ready whenever needed.

In a nutshell, the SPARQL query will identify whether the simple changes that comprise the complex one have been detected. To determine that, it will search in the special named graph \texttt{<detected\_changes>}, which contains all the detected changes, looking for instances of the appropriate simple changes that have been detected for the correct versions. Additionally, it will verify that all related filters are true and associate the parameters of the complex/simple changes.

For example, the SPARQL query for the complex change \textit{Label\_Obsolete} in Figure 10b will have the form:

\begin{verbatim}
SELECT ?v1, ?v2, ?v3 FROM <detected\_changes> WHERE {
    # parameters coupling
    ?anl_n1 anl_v1 ?v1.
    ?anl_n2 anl_v2 ?v2.
    ?lo_n1 refers\_to ?anl_n1.
    ?lo_n2 refers\_to ?anl_n2.
    ?mo_n1 mo_v1 ?v3.
    ?lo_n3 refers\_to ?mo_n1.
    # denote that simple changes refer on the same versions comparison
    ?anl anl\_p1 ?anl_n1.
    ?anl anl\_p2 ?anl_n2.
    ?anl old\_version ?ov.
    ?anl new\_version ?nv.
    ?mo mo\_p1 ?mo_n1.
    ?mo old\_version ?ov.
    ?mo new\_version ?nv.
    FILTER(?lo\_n1 = ?lo\_n3).
} 
\end{verbatim}

When the above query is about to be executed, the variables \texttt{?ov}, \texttt{?nv} which refer on the old and new version respectively will be associated with specific DIACHRON entity versions. This will be implemented by appending two \texttt{FILTER} statements i.e., \texttt{FILTER(?ov = "V1")}, \texttt{FILTER(?nv = "V2")}, where \texttt{V1}, \texttt{V2} are the old and new versions respectively (note that these are known at the execution time of the query, but not during the creation of the complex change).

If the WHERE clause of the above query returns a non-empty result, this means that one or more complex changes of the corresponding type are detectable, so they should be recorded as such. For efficiency, the above
SPARQL query will in fact be a SPARQL INSERT statement; if the WHERE clause returns one or more detected complex changes of said type, the INSERT will correctly create the appropriate triples at the data level to represent said change.

For the representation of the detected complex changes we follow the same approach as in the previous change types by inserting the appropriate instances in the ontology of changes for the corresponding detected changes, as well as the values of the corresponding parameters. For example, in Figure 11 we show the instances which are inserted in the case of the detection of \textit{Label} \_\textit{Obsolete} (not all of them are shown for clarity).

![Figure 11: Complex Change Detection](image)

### 2.3 Detecting Changes

In this section we will show how we can detect the changes described in previous sections. The detection process is independent from the data model which is adopted by the data pilots (RDF, relational or multidimensional) because the pilots’ datasets are mapped and imported into the DIACHRON repository organised into DIACHRON entities under the DIACHRON data model [35], and the change detection service will operate upon said model.

In a nutshell, change detection will consist of appropriately defined SPARQL queries (one per defined change), run over the DIACHRON repository, which will detect whether instances of said change exist; if so, these will be recorded in the repository, as explained in Section 2.2 above. More details are given below.

#### 2.3.1 Change Detection Module

In this subsection, we give the general overview of the Change Detection module. In general, we can discriminate two modes under which change detection will be called. The first case is when a new DIACHRON entity version is ingested into the DIACHRON archive. In this case, all changes (including low-level, basic, simple and complex) have to be detected, and the archiving service should call the change detection service to perform this detection.

The second case appears when the user edits the list of complex changes. In this case we have three specific subcases:
1. When a new complex change is added to the list of complex changes, it might happen that the new complex change appears in some of the previously detected pairs of versions. As it will be inefficient to rerun all previous detections to check this, we allow the user to perform an on-demand detection of this complex change (only) for any specific pair of versions.

2. When a new complex change is deleted from the list of complex changes, all associated detections have to be deleted too, because said complex change is no longer relevant. This is performed automatically whenever a complex change is deleted, so the respective algorithm will be embedded in the corresponding routine for deleting complex changes (details are trivial and omitted).

3. When a complex change is edited, we need to do both of the above, i.e., delete all the previous detections (which may no longer be relevant) and perform new detections. As before, the first step (deleting old detections) is done automatically when the complex change is deleted, whereas the second step (re-running the detections to check whether the new definition of the complex change is detectable between any pair) is performed on-demand.

Algorithm 1 describes the change detection module covering both modes of operation. The module requires as parameters the entity versions \(V_1, V_2\) and a flag \((\text{ingest})\) which denotes whether the routine is called because a new dataset version is ingested or not; the fourth parameter \((\text{CCs})\) contains a list of complex changes that need to be detected, and is relevant only under the second mode of operation (because under the first mode, all complex changes will be detected, by default).

If we have a new entity version \((\text{ingest} = \text{true})\), we have to detect all types of changes between the two versions. Thus, we first compute the low-level delta (line 2 – cf. Subsection 2.3.2 and Algorithm 2). Then, we fetch the set of all simple changes (line 3) and call \(\text{DetectSimpleChanges}\) to detect the simple changes (line 4 – cf. Subsection 2.3.4 and Algorithm 3). Note that \(\text{DetectSimpleChanges}\) will also detect basic ones. In line 5, the routine \(\text{fetchComplexChanges}\) is called to return all relevant complex changes. Note that this will overwrite the value of the \(\text{CCs}\) parameter, but this is intentional because under this mode of operation the detection of complex changes should take into account all complex changes. Finally, complex changes are detected and put in the repository in line 7 through \(\text{DetectComplexChanges}\) (cf. Algorithm 4 and the related discussion in Subsection 2.3.5).

Under the second mode of operation, i.e., when the user wants to detect complex changes that were defined after the original detection took place, only line 7 will be executed, using the \(\text{CCs}\) that were given in the input. Regarding this mode of operation, we are also considering a solution to allow the system to determine automatically which are the “relevant” complex changes to be detected, using timestamps: if a complex change was created after the original detection took place, then it will be considered (and put in \(\text{CCs}\)), otherwise it will not be considered. This solution is not presented here.

Algorithm 1 change_detection\((V_1,V_2,\text{ingest},\text{CCs})\)

\[
\begin{align*}
1: & \quad \text{if } \text{ingest} = \text{true} \text{ then} \\
2: & \quad \Delta := \text{compLLChanges}(V_1, V_2) \\
3: & \quad \text{SCs} := \text{fetchSimpleChanges}() \\
4: & \quad \text{DetectSimpleChanges}(V_1, V_2, \text{SCs}) \\
5: & \quad \text{CCs} := \text{fetchComplexChanges}() \\
6: & \quad \text{end if} \\
7: & \quad \text{DetectComplexChanges}(V_1, V_2, \text{CCs})
\end{align*}
\]

2.3.2 Low-level Changes Detection

The first step in the full change detection process (under the standard mode of operation) is to identify the low-level changes between the entity versions which are going to be compared, \(\Delta(V_1, V_2)\) (see also [34]). This is easy to do using appropriate SPARQL queries to create the corresponding sets of added and deleted triples; these queries are as follows:
• Added Triples:
  SELECT ?s ?p ?o WHERE {
  GRAPH <V2> { ?s ?p ?o }
  FILTER NOT EXISTS { GRAPH <V1> {?s ?p ?o } }
  }

• Deleted Triples:
  SELECT ?s ?p ?o WHERE {
  GRAPH <V1> { ?s ?p ?o }
  FILTER NOT EXISTS { GRAPH <V2> {?s ?p ?o } }
  }

These triples are put in the named graphs < added > and < deleted > respectively for use by subsequent queries; this can be made more efficiently by changing the above queries into SPARQL INSERT statements. Also note that < added > and < deleted > should be cleared before the detection, to clear any previous detection results.

2.3.3 Basic Changes Detection

The low-level changes are used during the detection of the basic and the simple ones. Regarding basic changes, recall that they are only relevant to guarantee the completeness of the detection. For this reason, Algorithm 1 does not explicitly call a routine for detecting basic changes; instead, it detects simple ones only (line 4). As we will see later (Algorithm 3 and Subsection 2.3.4), the algorithm for the detection of simple changes will call the routine for the detection of basic changes, and this will be done only if the detection of simple changes leaves some low-level changes not associated with any simple change. The algorithm for detecting basic changes for these “remnants” is described here.

The detection algorithm for basic changes considers the changes described in Appendix A. The general idea is that basic changes are detected one by one in line 5 by executing the appropriate SPARQL query (see also Appendix A). Detection takes place in random order; the order does not matter thanks to the unambiguity property of basic changes – see [34]. Once a basic change has been detected, we can safely eliminate from the low-level delta the triples that are involved in its detection, because they have already been “used” during the detection process (line 8). The elimination of triples from the low-level delta should be seen as a consequence of the detection of a specific change and will be called consumption, and it is safe thanks to the unambiguity property of basic changes. The process continues until all the low-level changes which were detected in the low-level delta have been consumed (see lines 2-4).

The SPARQL query used for the detection of basic changes is easy to construct from the definition of the change; the full list appears in Appendix A. For example, the SPARQL query related to the basic change \textit{Add\_Superclass}(a,b) in Appendix A is:

SELECT ?a, ?b WHERE {
  GRAPH <added> {?a rdfs:subClassOf ?b.
  FILTER (?b != rdfs:Resource) }
  GRAPH <V2> {?a rdf:type rdfs:Class.
  ?a rdfs:subClassOf rdfs:Resource.}
}

In the above query we search for all the added class subsumption relations; each result row (say \((a1, b1)\)) contains the parameters of a specific detected basic change of the required type (in our case, \textit{Add\_Superclass}(a1, b1)).

As already mentioned, this query is run in the context of Algorithm 2 whose input is a delta (\(\Delta\)) and a set of basic changes to consider during the detection process (\(BC\_Def\)). Normally, \(BC\_Def\) will contain all the basic changes shown in Appendix A.

In line 5 the SPARQL query for a specific basic change \(bcd\) is applied upon the delta \(\Delta\) and a set of parameter lists (which correspond to the detected instances of \(bcd\)) is created. Next, the algorithm spans the result set of the query (\(ParamLists\)) and creates the corresponding basic change instances in line 7. For example, if the algorithm
Algorithm 2 DetectBasicChanges($\Delta, BC_{Def}$)

1: for all basic change definitions $bcd \in BC_{Def}$ do
2: if $\Delta$ is empty then
3: break
4: end if
5: ParamLists := applySPARQL($bcd, \Delta$)
6: for all Parameter lists $list \in ParamLists$ do
7: $bc := createBasicChange(bcd, list)$
8: ConsumeLLChanges($bc$)
9: updateChangesOntology($bc$)
10: end for
11: end for

applies the SPARQL query for the change Add_Superclass, then a potential result row in ParamLists (e.g., $(a_1, b_1)$) would create the basic change instance Add_Superclass$(a_1, b_1)$. Then, the method ConsumeLLChanges (line 8) is used to remove from the delta the low-level changes that are consumed due to the detection of the basic change $bc$.

Finally, the detected change is inserted in the ontology of changes as a set of appropriate property instances as described in Subsection 2.2 (line 9). For example, if the basic change Add_Superclass$(a_1, b_1)$ is detected, based on Figures 6, 7, we have to insert the following triples in the repository (for brevity, we omit the rdf:type triples instantiating the detected change and its parameters under their corresponding classes):

$\langle asc_1, asc_p1, asc_n1 \rangle$, $\langle asc_1, asc_p2, asc_n2 \rangle$, $\langle asc_n1, asc_v1, a_1 \rangle$, $\langle asc_n2, asc_v2, b_1 \rangle$.

The detection process finishes when all the low-level changes have been consumed. This is guaranteed to happen, due to the completeness property [34]. If, however, the set $BC_{Def}$ contains a subset of the basic changes shown in Appendix A, then it is possible that the routine will run through all the basic changes and still leave some low-level changes unconsumed; this would cause it to end without detecting all changes that actually happened. For this reason, it is strongly recommended that $BC_{Def}$ is always populated with its default value, namely all the basic changes in Appendix A.

For efficiency, we plan to implement the above algorithm as a SPARQL update statement, which will perform the above actions (namely, the detection of the change and its introduction in the repository) in one blow per basic change. Due to unambiguity and completeness, the consumption part can be performed after the end of the process.

2.3.4 Simple Changes Detection

For the detection of simple changes we use an approach similar to the one followed in the basic changes; however, note that simple changes have some extra features (i.e., priority, presence, condition) which do not exist in the basic changes and need to be considered in the algorithm. The detection algorithm for the simple changes appears in Algorithm 3 and is called from the main routine of the change detection service (Algorithm 1). Details for this algorithm are given in this subsection.

As before, the idea is that we run the corresponding SPARQL queries for each simple change (line 5 in Algorithm 3), consume the respective low-level changes (line 8) and update the repository with the newly detected changes (line 9). Despite the same general idea, there are some differences that will be analysed below.

In more detail, Algorithm 3 requires as input the low-level delta $\Delta$ and a set of simple changes which are going to be detected ($SC_{Def}$). This by default will be all the simple changes understood by the DIACHRON language of changes, but one could restrict it, e.g., to only the changes related to a specific pilot or DIACHRON entity.

The first step of the algorithm is to run through all the simple changes in $SC_{Def}$. At this point appears the first major difference with respect to Algorithm 2, because here the order matters. As mentioned before, simple changes will not necessarily be unambiguous, so the notion of priority has been introduced to prioritize their detection. To incorporate this feature in the algorithm the set of simple changes in $SC_{Def}$ must be scanned taking into account
the priority, i.e., the main FOR loop of Algorithm 3 will not consider the changes in a random order, but in the order imposed by their priority (priority 1 comes first, followed by priority 2 etc).

In line 5, the query for detecting each simple change is run. Defining the respective SPARQL query for each simple change, given the formal definition of simple changes as required in Subsection 2.1 (and also Subsection B.3.1 in Appendix B) is straightforward, and is performed at design time. As an example, the simple change Add new label(A, “label”) would be detected using the following SPARQL query:

```sparql
SELECT ?a, ?lab WHERE {
  GRAPH <added> {
    ?r1 diachron:hasAttribute ?ratt1.
    ?ratt1 diachron:predicate rdfs:label.
    ?ratt1 diachron:predicate ?lab.
  }
}
```

Note that in Add new label, all low-level changes are mandatory (cf. Subsection 2.1); in case we had optional changes, we would simply use the OPTIONAL construct of SPARQL, instead of join, for the respective triples. Conditions can also be straightforwardly added, as in the case of basic changes.

**Algorithm 3 DetectSimpleChanges(Δ, SC Def)**

1: for all simple change definitions scd ∈ SC Def do
2: if Δ is empty then
3: break
4: end if
5: ParamLists := applySPARQL(scd, Δ)
6: for all Parameter lists list ∈ ParamLists do
7: sc := createSimpleChange(scd, list)
8: ConsumeLLChanges(sc)
9: updateChangesOntology(sc)
10: end for
11: end for
12: BC Def := the set of all basic changes
13: DetectBasicChanges(Δ, BC Def)

As with the basic changes, all results returned by the query correspond to detected instances of said simple change. Thus, in line 7 we record the detected simple changes, and, as in the case of Algorithm 2, we use ConsumeLLChanges to consume the low-level changes related to said simple change (line 8). For example, if the algorithm applies the SPARQL query for the change Add new label, and gets the result (a1, “New Label”), then it will create the simple change instance Add new label(a1, “New Label”) and remove the corresponding changes in line 8.

Then, in line 9, the repository is updated with the triples that record the detection of the corresponding simple change (see Subsection 2.2). For example, if the simple change Add new label(a1, “New Label”) is detected, based on Figures 8b, 9 the triples that will be inserted in the ontology will be (as usual, instantiation triples are omitted):

(\(anl_1, anl_p1, anl_n1\)),
(\(anl_1, anl_p2, anl_n2\)),
(\(anl_n1, anl_y1, a1\)),
(\(anl_n2, anl_z2, "New Label"\)).

The process ends when either all the low-level changes have been consumed or when all simple changes in SC Def have been considered. As simple changes are not necessarily complete, it is possible that they are unable to consume all the low-level changes. In such cases the remaining low-level changes can be consumed by the basic changes we have defined in Appendix A which are defined to be complete. To achieve this effect, lines 12-13 call DetectBasicChanges to guarantee that (note that if all low-level changes have been consumed already, DetectBasicChanges will detect nothing).

As with basic changes, we plan to implement the above algorithm as a SPARQL update statement, which will
perform the detection of the change and its introduction in the repository in one blow per simple change. Due to the fact that changes of different priority might be ambiguous, the consumption part should be performed after all the changes of one priority level have been considered.

2.3.5 Complex Changes Detection

The detection of complex changes is more tricky because it requires the detection of changes whose specifications are not known at design time. Furthermore, the notion of consumption is void for complex changes, and complex changes do not rely on low-level ones, but on simple changes; therefore, their detection should access the repository of detected (simple) changes, rather than the low-level changes. In this subsection, we detail how complex changes are detected (see also Algorithm 4). Note that this algorithm will be called under both modes of operation of the Change Detection service; these modes are not relevant for Algorithm 4, because its parameters include instructions on the complex changes that it will try to detect.

Algorithm 4 requires as input the two entity versions which are going to be compared \((V_1, V_2)\) and a set of complex changes \((CC_{Def})\); the latter set must be a subset of all the complex changes whose definitions are already stored in the ontology of changes (see Subsection 2.2).

**Algorithm 4** DetectComplexChanges\((V_1, V_2, CC_{Def})\)

1. **for all** complex change definitions \(ccd \in CC_{Def}\) **do**
2. \(sparqlStmt := fetchSPARQLStmt(V_1, V_2, ccd)\)
3. \(updateSPARQLStmt(sparqlStmt, V_1, V_2)\)
4. \(ParamLists := applySPARQL(sparqlStmt)\)
5. **for all** Parameter lists \(list \in ParamLists\) **do**
6. \(cc := createComplexChange(cc, list)\)
7. \(updateChangesOntology(cc)\)
8. **end for**
9. **end for**

Detecting complex changes relies on the information found in the ontology of changes. In particular, for each complex change which is included in \(CC_{Def}\), we need to fetch its corresponding SPARQL (which is accessible in the ontology of changes through the property \(sparql\) – cf. Figure 10a); this is done in line 2 using the routine \(fetchSPARQLStmt\). As mentioned in Subsection 2.2 this SPARQL query is used to associate the complex change (and parameters) with its related simple changes (and parameters). The returned SPARQL query needs to be enhanced using the information on versions, as detailed also in Subsection 2.2; this is done in line 3.

After that, we are ready to apply the query to the DIACHRON repository to get the detected complex changes (line 4); unlike the case of basic and simple changes though, this time the part of the repository storing the detected simple changes is accessed, rather than the low-level delta. As usual, each query result corresponds to one detected instance of the corresponding complex change. Care should be taken with respect to the visibility of parameters. For example, consider the defined complex change \(Label_{Obsolete}(lo_1, lo_n2, lo_n3)\) which appears in Figure 10b: this change must be presented as \(Label_{Obsolete}(lo_n1, lo_n2)\) to the user, because the third parameter \((lo_n3)\) is set to invisible (via visible = false).

Finally, all the detected complex changes are stored in the repository (line 7), as detailed in Subsection 2.2. For example, if the change \(Label_{Obsolete}(lo_1, lo_n2, lo_n3)\) is detected, then, according to Figure 11, the triples which will be inserted in the repository will be (as usual, instantiation triples are omitted):

\[(lo, lo_p1, lo_o1),\]
\[(lo, lo_p2, lo_o2),\]
\[(lo, lo_p3, lo_o3),\]
\[
\ldots\]
\[(lo_o1, refersTo, mo_n1),\]
\[(lo_o2, refersTo, mo_n2),\]
\[
\ldots\]
The algorithm terminates when all the complex changes have been examined. In this case, we have no other terminating check, as there is no notion of consumption.
3 Change Monitoring and Propagation

3.1 Introduction

The monitoring and propagation service provides publish-subscribe functionality for creating change monitoring tasks and notifications. A monitoring task is a process configured by the user where a dataset is periodically inspected for changes. Change detection is performed by employing the Change Detection module functionality. The user may configure the task by declaring rules on the types of changes and the dataset resources she wishes to monitor. Other users subscribe to one or more monitoring tasks so that notifications for detected changes are propagated to them.

The datasets can be either datasets that already exist in the DIACHRON platform as part of the archive or they can be datasets that will be crawled/downloaded, using the platform’s Crawling module, for the purposes of this task. In the latter case the platform will store these datasets to the archive so as to be able to provide the required functionality. By this way, these datasets do not necessarily become part of the archive but only versions or parts of them needed for the monitoring tasks will be stored. Data storage is provided by the Data Access-Query Engine module.

A single monitoring task for a dataset can serve multiple users. Once it has been created by a user of the platform, other users could also subscribe to it. Monitoring tasks can be reused and thus changes will be propagated to them without having to recreate a monitoring task for the specific dataset. Hence, DIACHRON platform can act as a proxy for users that are interested in external LOD datasets and their evolution.

Once changes are detected, the propagation service will match changes to monitoring tasks based on the configuration parameters and subscribers will be appropriately notified. Change notifications will be propagated to users by various ways such as email notifications or ATOM feeds or similar syndication technologies. In this context, we will develop a scalable mechanism in terms of the number of tasks and changes processed, and the number of subscribers and notifications they receive.

3.2 Related Work

Publish-subscribe communication paradigm is receiving increased attention for the loosely coupled form of interaction it provides in large scale settings. Subscribers express their interest on an event, or a pattern of events, and are subsequently notified of any event generated by publishers, which matches their registered interest. An event is asynchronously propagated to all subscribers that registered interest in that event.

The strength of this interaction style lies in the full decoupling in time, space and synchronization between publishers and subscribers, as depicted in Figure 12. Time decoupling means that the interacting parties do not need to be actively participating in the interaction at the same time. Space decoupling means that interacting parties do not need to know each other. Synchronization decoupling means that publishers are not blocked while producing events and subscribers can get asynchronously notified (through a callback) while performing some concurrent activity.

Publish-subscribe systems differ on a number of fundamental characteristics. In next paragraphs, we discuss design options and provide categorizations of publish/subscribe systems.

Based on the subscription scheme followed, the main flavors of publish-subscribe systems are topic-based, content-based and type-based [19, 8]. In a topic-based subscription scheme, notifications are grouped into logical channels, and subscribers declare their interest to a particular topic. Note that a topic-based scheme can be extended to provide hierarchical organization of the topic space instead of a simple flat structure. In case of content-based scheme, subscribers specify conditions over the content of notifications they want to receive, composing a filter in a query form. In type-based scheme, events are actually objects of a specific type, which can encapsulate attributes and methods.

Regarding the system architecture, publish-subscribe systems can be classified into the general categories of client-server and peer-to-peer [30, 8]. In client-server, clients may act as publishers, subscribers, or both, while servers (each one of them is called broker) may be related with several kinds of topologies, like star topology (centralized server), hierarchical, ring or irregular polygon topology. In peer-to-peer, all nodes are equal, and
therefore each node may act as publisher, subscriber, root of a multicast tree, internal node of a multicast tree, or any reasonable combination of thereof.

Matching is performed by publish-subscribe systems in order to determine whether dispatching an event to a subscriber. In topic-based systems, matching is reduced to a simple table lookup. However, in content-based systems it is a fundamental issue for the overall performance. Techniques for efficient matching can be grouped in two main categories: predicate indexing algorithms and testing network algorithms. Predicate indexing algorithms are structured in two phases: first, subscriptions are decomposed into elementary constraints and it is determined which constraints are satisfied by the notification; next, the results of the first phase are used to determine the filters in which all constraints match the event. Testing network algorithms are based on a preprocessing of the set of subscriptions that builds a data structure composed by nodes representing the constraints in each filter. The structure is traversed in a second phase of the algorithm, by matching the event against each constraint. An event matches a filter when the data structure is completely traversed by it [8].

As for the notifications delivery mechanisms [19], push-based versus pull-based can be considered. Pull-based mechanism involves that a request is received at the server, which then locates the information and returns it to the client. Push-based involves sending events to a client in advance of any request. Moreover, delivery can be characterized as aperiodic, meaning that it is event driven (an event might be a client request for pull, or an event publication for push), and 1-to-N, meaning that multiple subscribers can receive copies a specific published event.

There are many open source and commercial implementations of the publish/subscribe paradigm. For example, Apache Kafka[7] is an open source publish/subscribe system, implemented as a distributed, partitioned, replicated commit log service, while Pubsubhubbub[8] is an open webhook-based publish/subscribe protocol.

It is worth noting that change monitoring and notification is primarily supported in traditional RDBMS through triggers. Also, many commercial and open source RDBMS provide further mechanisms for monitoring changes and sending related messages. For example, Oracle’s Database Change Notification[9] is a feature that enables client applications to register queries with the database and receive notifications in response to DML or DDL changes on the objects associated with the queries when the respective transaction commits. Also, query notifications[10] were introduced in SQL Server 2005 and SQL Server Native Client, based on Service Broker [11]. Query notifications

---

3[http://docs.oracle.com/cd/B19306_01/appdev.102/b14251/adfns_dcn.htm](http://docs.oracle.com/cd/B19306_01/appdev.102/b14251/adfns_dcn.htm)

---

Figure 12: Space, Time and Synchronization Decoupling [19]
allow requesting notification within a specified time-out period, when the underlying data of a query changes. Notifications are delivered through a Service Broker queue that applications may poll for available notifications. PostgreSQL provides listen/notify functionality\(^\text{12}\), so that each client application that has registered as a listener to a notification channel is notified, whenever the command NOTIFY channel is invoked. For example, NOTIFY command may be used to signal the occurrence of changes to a particular table, or a useful programming technique is to put this operation in a rule that is triggered by table updates, so that notifications are made automatically when the table changes. Finally, in Sybase Unwired Platform\(^\text{13}\) a data change notification mechanism is also supported. This is an update mechanism that reconciles enterprise information system data changes with Unwired Server over HTTP(S) connection, which pushes notification messages for changes to clients (mobile devices).

A change detection framework for Linked Data sources that informs data consumers about events (create, remove, move, update) that occur on data is presented in \cite{37}. It is based on an indexing infrastructure, where a monitor accesses considered Linked Data sources, extracts feature vectors and stores them to an index. The monitor detects changes on resources based on this index. These events are written to a central event log and notifications are sent to registered applications.

3.3 Requirements

This section describes the use case scenarios supported by the monitoring and propagation service. The user of the service can create and manage monitoring tasks, subscribe and unsubscribe from monitoring tasks and receive change notifications.

3.3.1 Create a Monitoring Task

Description: The user provides the monitoring task configuration, which consists of the following parameters:

- the id of the dataset that is to be monitored,
- the monitoring period during which the dataset is to be monitored for changes,
- the monitoring frequency over which the dataset is to be probed for changes,
- the change types of interest which are expected to be reported, following the classification proposed in Section 2 of the current deliverable,
- the dataset part that changes of interest may affect, corresponding to the diachronic resources as defined in \cite{35}.

Also, the user declares whether the monitoring task will start running immediately. The monitoring and propagation service returns a monitoring task id.

Option: If the configuration is invalid, the service responds with an “invalid configuration” message. Note that a given configuration is invalid, when it is identical to an already existing monitoring task configuration, or when a wrong dataset id is given, or the dataset part definition evaluates into an empty dataset.

3.3.2 Modify a Monitoring Task

Description: The user provides new configuration about a monitoring task. The parameters which may be altered are: the monitoring period, the monitoring frequency, the change types and the dataset part of interest. The monitoring and propagation service replies with a “verification” message.

Option 1: If new configuration is invalid, the service responds with an “invalid configuration” message.

Option 2: If the user is not the one who created or started the monitoring task (may be just a subscriber or not), the service replies with an “unauthorized user” message.

\(^{12}\)http://www.postgresql.org/docs/9.0/static/sql-notify.html

3.3.3 Delete a Monitoring Task
Description: The user requests a monitoring task configuration, to be deleted. The monitoring and propagation service replies with a “verification” message.
   Option: If the user is not the one who created or started the monitoring task (may be just a subscriber or not), the service replies with an “unauthorized user” message.

3.3.4 Start a Monitoring Task
Description: The user requests a monitoring task to start running, given its configuration. The monitoring and propagation service returns the time when it started.
   Option: If the monitoring task is already running, the monitoring and propagation module responds with an “already started” message.

3.3.5 Stop a Monitoring Task
Description: The user requests a monitoring task to stop running. The monitoring and propagation service returns the time when it stopped.
   Option 1: If there is no such monitoring task running or being paused, the service replies with an “already stopped” message.
   Option 2: If the user that did the request is not the one who started the monitoring task (maybe she is just subscriber or not), the service replies with an “unauthorized user” message.

3.3.6 Pause a Monitoring Task
Description: The user requests a monitoring task to stop running for a specific period, and then it resumes. The service returns the time when it paused.
   Option 1: If there is no such monitoring task running or being already paused, the service replies with an “already stopped” message.
   Option 2: If there is no such monitoring task running or being already stopped, the service replies with an “already paused” message.
   Option 3: If the user that did the request is not the one who started the monitoring task (maybe she is just subscriber or not), the service replies with an “unauthorized user” message.

3.3.7 Request Monitoring Task Configuration
Description: The user requests a monitoring task configuration and the service returns all configuration parameters.

3.3.8 List Monitoring Tasks
Description: The user requests all created monitoring tasks to be listed and the service returns all monitoring task ids.

3.3.9 Subscribe to a Monitoring Task
Description: The user makes a subscription request with the following parameters:
   - the monitoring task id that she wants to subscribe,
   - the subscription period during which she wants to get notifications,
   - the notification type she wants to receive (email, ATOM feeds, etc).

The service returns a subscription id.
3.3.10 Unsubscribe from a Monitoring Task

Description: The user makes an unsubscribe request from a monitoring task. The service replies with a “verification” message.

3.3.11 Propagate Notifications

Description: The user receives notifications from a monitoring task that she has subscribed to, in the requested form.

3.4 Architecture

In this section, we present the monitoring and propagation service architecture, as depicted in Figure 13. We introduce its basic components, their functionality and how they interact with each other.

The service architecture has two layers: the integration layer and the business logic layer. Integration layer allows the service to interact with the service consumers outside of the platform, like end users and applications, as well as other DIACHRON components. It actually represents the integration layer as described in the overall system architecture in [35]. Business logic layer is the main part of the service implementing its functionality.

The main components of the service architecture are the following:

- the Web UI and HTTP API, which allow the communication of the service consumers with it,
- the Monitoring and Propagation API, which offers a suite of operations over the service,
- the Mediator, which allows the communication with other DIACHRON components,
- the Monitoring Manager, which coordinates the monitoring tasks,
- the Propagation Manager, which coordinates subscriptions and notification propagation,
- the Configuration Repository, which provides storage for monitoring tasks configurations and subscriptions.

**Monitoring and Propagation API** provides a set of operations that the service provides to the end users. It allows users to create, manage, subscribe and unsubscribe from monitoring tasks in order to start or stop receiving notifications for changes. The monitoring and propagation API communicates with monitoring and propagation managers, which provide the core module functionality. It sends requests regarding monitoring tasks and subscriptions respectively and receives outputs or messages.

**Mediator** permits the communication with other DIACHRON components, which are: the Change Detection module, for detecting the changes that the user wants to be notified for, the Data Access-Query Engine module, for retrieving dataset versions from the archive, and the Crawling module, for downloading datasets from the web in order to be monitored. In this way, it helps monitoring tasks to provide the monitoring functionality. Also, it receives notifications when a new dataset version is inserted into DIACHRON Archive, so that the monitoring manager triggers the related monitoring tasks.

**Monitoring Manager** coordinates the monitoring tasks. It stores, retrieves, modifies and deletes monitoring task configuration parameters. These parameters constitute a description of a monitoring task and are stored into the Configuration Repository. Specifically, they specify the dataset, the time period and frequency over which it is going to be monitored, and the changes and diachronic resources of interest.

Given a set of monitoring task configuration parameters, the monitoring manager can start a monitoring task. In this way, it creates a process that probes a dataset for changes and publishes them to the propagation manager. A monitoring task may act on demand or periodically. Assume that a dataset is already stored into DIACHRON Archive. When a new dataset version is issued and the DIACHRON Archive is updated with this version, all monitoring tasks defined on this dataset can be triggered. This is an on demand call of a monitoring task. On the other hand, a monitoring task may be called periodically, over a given period of time. This is more likely to be the case when a monitored dataset is being crawled from the web and it is not already archived within DIACHRON
The monitoring manager can also pause or stop the execution of a monitoring task, and list defined monitoring tasks.

**Propagation Manager** coordinates subscriptions and efficiently propagates notifications in the requested form. Specifically, it receives requests for subscribe/unsubscribe from a monitoring task, it stores subscription options in the Configuration Repository, it matches published changes with subscriptions and propagates accordingly the notifications. Notification delivery policy can be either pull or push.

Each time change detection runs during a monitoring task, a notification containing all the detected changes is propagated. Notifications have a uniform structure for their content. Each notification includes a header and a body message containing the detected changes described according to the structure defined in Section 2.

**Configuration Repository** provides storage for monitoring tasks configuration parameters and subscription options.

### 3.5 Workflows

Below, we place some basic workflow diagrams depicting monitoring and propagation service operation during some of the use case scenarios defined in Section 3.3. Note that the workflow diagrams for pausing a monitoring task and unsubscribe from a monitoring task are omitted as they resemble with the ones for stopping and subscribing a monitoring task respectively, while workflows for requesting monitoring task configuration and listing...
monitoring tasks are considered as straightforward. Figure 20 depicts notification propagation for a monitoring task running on demand, after a notification of a new dataset version update to archive. Finally, Figure 21 depicts notification propagation for a monitoring task running periodically, for a dataset being crawled from the web for the purposes of this task (the figure depicts the service operation during one iteration).

Figure 14: Workflow Diagram for Creating a Monitoring Task
Figure 15: Workflow Diagram for Modifying a Monitoring Task

Figure 16: Workflow Diagram for Deleting a Monitoring Task
Figure 17: Workflow Diagram for Starting a Monitoring Task

Figure 18: Workflow Diagram for Stopping a Monitoring Task
Figure 19: Workflow Diagram for Subscribing to a Monitoring Task

Figure 20: Workflow Diagram for Propagating Notifications, when the Monitoring Task Runs on Demand
Figure 21: Workflow Diagram for Propagating Notifications, when the Monitoring Task Runs Periodically
4 Repairing and Cleaning

4.1 Repairing

4.1.1 Introduction

Many of the datasets found in the Data Web are associated with adequate schemas and integrity constraints (generic or application/domain specific) [26, 32, 41]; these are intended to describe the corresponding dataset and capture different quality aspects related to the data and, thus, they are required to hold for the corresponding dataset. For example, one may want to impose functionality properties (or primary key constraints [26]), cardinality constraints [32], disjointness constraints [32] and others. An interesting example of such a constrained dataset is the recent PROV data model\(^\text{14}\), whose aim is to enable the inter-operable interchange of provenance information in heterogeneous environments such as the Web, and is associated with several integrity constraints\(^\text{15}\).

Unfortunately, invalidities, i.e., violations of integrity constraints appear often. For example, invalidities may arise as scientists, acting as curators, have to constantly agree on the common knowledge to represent and share in their research field. Changes performed in the light of new experimental evidence and observations, due to revisions in the dataset’s intended usage, or even to correct erroneous conceptualizations, may also cause invalidities. In addition, the interconnection aspect may give rise to invalidities, when changes get propagated from related remote datasets, even when invalidities could be locally prevented. Furthermore, invalidities may be caused by changes in the constraints themselves [39]. In DIACHRON, we will propose a generic and personalized repairing framework for assisting curators in the arduous task of enforcing integrity constraints in large datasets, which is mostly enforced nowadays manually.

The repairing service of DIACHRON (which is discussed in this section) will deal with the problem of identifying and resolving logical invalidities in datasets, i.e., the violation of constraints like disjointness, functionality constraints etc, taking into account logical inference. Other types of constraints, such as constraints related to literals (e.g., values being within a certain range, strings having a certain form etc) are dealt with by the cleaning service (discussed in Section 4.2).

As regards to the repairing service, we will consider constraints that can be expressed in the language DL-Lite\(_A\) [12], and provide an efficient methodology for identifying invalidities (diagnosis), as well as for resolving them (repairing) in a manner that has the least impact (in terms of lost knowledge) from the dataset, according to the principles set out in earlier works [4, 14, 39].

The choice of DL-Lite\(_A\) was motivated by the fact that this language is arguably rich enough to express several useful types of integrity constraints, while at the same time being efficient [12]. Efficiency is a very important factor in the context of DIACHRON, having in mind the size of the datasets that we will be dealing with. DL-Lite\(_A\) provides query answering (which is the foundation of the diagnosis component) in LOGSPACE with respect to the size of the data, while providing many of the main features necessary to express conceptual models such as UML class diagrams and ER diagrams [11]. It is designed so that data that is stored in a standard database system can be queried through an ontology via a simple rewriting mechanism.

The integrity constraints that can be captured by DL-Lite\(_A\) are concept disjointness constraints, property disjointness constraints, property domain and range disjointness constraints, functionality constraints and disjointness between value-domains (see also Appendix C). Note that DL-Lite\(_A\) constraints allow the presence of interrelated violations, i.e., violations whose potential resolutions coincide; this implies that there are resolutions which resolve more than one violated constraints at the same time.

In the rest of this section, we provide some related work on the problems of diagnosis and repair for logical invalidities, an introduction to DL-Lite\(_A\), as well as details on our approach for addressing the above problems (diagnosis and repair) in a generic fashion, and in the context of DIACHRON.

\(^{14}\)http://www.w3.org/TR/prov-overview/

\(^{15}\)http://www.w3.org/TR/2013/REC-prov-constraints-20130430/
4.1.2 Related Work on Diagnosis and Repair

The problem of inconsistencies appearing in knowledge bases (KBs) can be tackled either by providing the ability to query inconsistent data and get consistent answers (Consistent Query Answering - CQA) [6], or by actually repairing the KB, which leads to a consistent version of it [22]. Both these approaches have attracted researchers’ attention, mostly in the context of relational databases and, lately, in the context of linked data and ontology languages as well.

In the context of relational databases, CQA has been studied in various works dealing with different classes of conjunctive queries and denial constraints, mainly key constraints ([15, 20, 23, 45]). These works underline the main advantages of using First-Order query rewriting for the validation of integrity constraints.

Regarding the repairing of inconsistent databases, different semantics have been studied, considering different kinds of constraints. For example, [15] studied the problem of repairing by allowing only tuple deletions and, in this way, resolving violations of denial constraints. Automatic repairing with respect to a set of conditional functional dependencies (CFDs) has been studied in [16]. Moreover, attribute-based repairs, that is repairs obtained by changing some attribute values in existing tuples, were studied in [44] and [10].

Regarding the context of linked data and the corresponding languages and technologies, there has been work on the topic of using ontology languages to encode integrity constraints (ICs) that must be checked over dataset instances. In [42], the authors present a way to integrate ICs in OWL and they show that IC validation can be reduced to query answering, for integrity constraints that fall into the SROIQ DL fragment. A similar approach has been followed in [32], for a different DL language (ALCHI). In [26], the presented approach integrates constraints that come from the relational world (primary-key, foreign-key) into RDF and provides a way to validate these constraints.

Integrity constraint validation (ICV) is also an important part of some of the current OWL reasoners. One of those reasoners, the one that is part of Stardog16, provides the ability to express and validate integrity constraints expressed in SPARQL, OWL or SWRL and combines this ability with OWL 2 Reasoning.

The approaches discussed above in the context of ontology languages differ significantly from the work in DIACHRON, as they only support validation of constraints and they don’t deal with the diagnosis of invalid data instances and the repair of such invalidities.

Previous work by FORTH [39] studied the problem of repairing in a more general setting, considering a much richer set of possible constraints than the one we are considering here. That work showed that the use of very expressive constraints creates scalability problems, as the problem becomes exponentially difficult with respect to the number of violations [39]. For this reason, we chose to support a less expressive language for the DIACHRON context, which, however, is still expressive enough to capture constraints that commonly appear in real-world datasets, and cover most of the requirements expressed by the DIACHRON pilots (see also Appendix D).

In recent years, there has also been some research on the problems of diagnosis and repair of invalidities in the context of Description Logics (DL) languages. Most of this work focuses on CQA for inconsistent knowledge bases, by using query rewriting techniques. To the best of our knowledge, little work has been performed on the problem of automatically computing a repair for inconsistent KBs.

Regarding CQA on DLs, [28, 29] deal with different variants of inconsistency-tolerant semantics to reach a good compromise between expressive power of the semantics and computational complexity of inconsistency-tolerant query answering. More precisely, these works come to the conclusion that there are inconsistency-tolerant semantics that are FOL-rewritable, and therefore give good complexity behaviour with respect to data complexity.

In the field of diagnosis for DL-Lite KBs, there has been some work regarding inconsistency checking. The DL-LiteA reasoner QUONTO [3] has the ability to check an ABox for inconsistencies with respect to a given TBox and to report the TBox axioms that were violated. However, it does not have the ability to return the ABox assertions that violate the specific TBox axioms. QUONTO is currently a part of Mastro system17.

To the best of our knowledge, the only work dealing with the automatic repairing of an invalid DL-Lite ABox, with respect to a TBox, appears in [31]; this is work in progress and is based on the inconsistency-tolerant semantics studied in [29].

---

16http://stardog.com/
17http://www.dis.uniroma1.it/~mastro/
4.1.3 Preliminaries

As explained above, the repairing process is about detecting and providing possible repairs of invalidities attributed to constraints of a purely logical nature (e.g., class $A$ is disjoint with class $B$). For the detection of such invalidities, a set of integrity constraints must be provided that will be diagnosed over a dataset instance. In order for the diagnosis process to be efficient, this set of integrity constraints must be expressed in an ontology language that, while being expressive enough to capture the main conceptual modelling formalisms, can preserve the tractability of query answering with respect to the size of data, which can rely on a standard DBMS.

A family of ontology languages that is able to satisfy the above prerequisites for the diagnosis and repairing process, is the DL-Lite family of Description Logics [13], on which OWL 2 QL\(^\text{18}\) is based. More precisely, the maximal languages of the DL-Lite language family that possess the above properties are DL-Lite$_R$, DL-Lite$_F$ and DL-Lite$_A$ [12]. Here, we focus on the latter (DL-Lite$_A$), which is the most expressive of the three. The landscape of the different fragments of the DL-Lite family according to the data complexity of query answering is illustrated in Figure 22.

![Figure 22: The DL-Lite Family and Relations](image)

The unconditional combination of the features of DL-Lite$_R$ and DL-Lite$_F$ leads to query answering that is \(\text{PTIME-HARD}\) with respect to the size of data. DL-Lite$_A$ comes to resolve the above problem (keeping the query answering in \(\text{LOGSPACE}\)), by combining the main features of both DL-Lite$_R$ and DL-Lite$_F$ and imposing a limitation on the use of functionality assertions in the TBox. This limitation in the TBox is that every functional role cannot be specialized by using it in the right-hand side of a role inclusion assertion.

In the following, we describe the specification of the DL-Lite$_A$ logic in more details. The other two languages (DL-Lite$_R$ and DL-Lite$_F$) follow a similar specification, which is omitted here (but can be found in [13]). However, all those three languages are covered by the diagnosis and repair process that we will explain in this section.

The notation used in the literature to provide the specification of the DL-Lite$_A$ logics is the following:

- $A$ denotes an atomic concept, $B$ a basic concept, $C$ a general concept, and $\top_C$ denotes the universal concept.
- $D$ denotes an atomic value-domain, $E$ denotes a basic value-domain, $F$ a general value-domain, and $\top_D$ the universal value-domain.

\(^{18}\)http://www.w3.org/TR/owl2-profiles/#OWL_2_QL
• P denotes an atomic role, Q a basic role, and R a general role.

• UC denotes an atomic concept attribute, and VC a general concept attribute.

• UR denotes an atomic role attribute, and VR a general role attribute.

Given a concept attribute UC (respectively a role attribute UR), we call the domain of UC (respectively UR), denoted by δ(UC) (respectively δ(UR)), the set of objects (respectively pairs of objects) that UC (respectively UR) relates to values, and we call range of UC (respectively UR), denoted by ρ(UC) (respectively ρ(UR)), the set of values that UC (respectively UR) relates to objects (respectively pairs of objects). Furthermore, we denote with δF(UC) (respectively δF(UR)) the set of objects (respectively of pairs of objects) that UC (respectively UR) relates to values in the value-domain F.19

The DL-LiteA language allows the following expressions:

1. Concept expressions:
   \[ B \rightarrow A \mid \exists Q \mid \delta(UC) \]
   \[ C \rightarrow \top \mid B \mid \neg B \mid \exists Q.C \mid \exists \delta_F(UC) \mid \exists \delta_F(UR) \mid \exists \delta_F(UR) \]

2. Value-domain expressions:
   \[ E \rightarrow D \mid \rho(UC) \mid \rho(UR) \]
   \[ F \rightarrow \top \mid E \mid \neg E \mid rdf>DataType \]

3. Role expressions:
   \[ Q \rightarrow P \mid P^- \mid \delta(UR) \mid \delta(UR)^- \]
   \[ R \rightarrow Q \mid \neg Q \mid \delta_F(UR) \mid \delta_F(UR)^- \]

4. Attribute expressions:
   \[ V_C \rightarrow U_C \mid \neg U_C \]
   \[ V_R \rightarrow U_R \mid \neg U_R \]

The assertions that can appear in a DL-LiteA TBox (ontology schema) are of the form:

\[ B \sqsubseteq C \] (concept inclusion assertion)
\[ Q \sqsubseteq R \] (role inclusion assertion)
\[ E \sqsubseteq F \] (value-domain inclusion assertion)
\[ U_C \sqsubseteq V_C \] (concept attribute inclusion assertion)
\[ U_R \sqsubseteq V_R \] (role attribute inclusion assertion)

(funct P) (role functionality assertion)
(funct P^-) (inverse role functionality assertion)
(funct UC) (concept attribute functionality assertion)
(funct UR) (role attribute functionality assertion)

The DL-LiteA TBox assertions described above can be trivially expressed in OWL syntax. This can be seen in [17]. For example, (funct P) can be expressed in OWL as FunctionalObjectProperty(P), using the functional-style syntax of OWL.

What makes DL-LiteA a good candidate for expressing integrity constraints in the context of DIACHRON diagnosis and repair, is the notion of FOL-Reducibility. This property captures the fact that one can reduce the

19This δ should not be confused with the δ used to indicate sets of changes in the previous sections. We use this notation here because it is the notation that has been mainly used in the related DL literature.
process of KB satisfiability check and query answering to evaluating a FOL (first-order logic) query over the ABox (dataset instance), considered as a database. That makes KB satisfiability check and query answering tractable (in LOGSPACE with respect to the data) [13].

Using the property of FOL-Reducibility, we can perform diagnosis on the data with respect to the underlying schema, by posing the integrity constraints expressed in DL-LiteA syntax as FOL queries over the database. For example, if the TBox contains the constraint $A \sqsubseteq \neg C$, we can check the data for invalidities based on this constraint, by executing the FOL query $q(x) \leftarrow A(x), C(x)$ over the database. The answers to this query represent inconsistencies between the TBox and the ABox and have to be repaired. The possible repairs in this case are the removal of either $A(x)$ or $C(x)$ from the database, or the removal of both, for each $x$ that is an answer to the query. Note that the latter (removing both) will not be considered in DIACHRON, as it violates the requirement for the repairing process to have a minimal impact (in terms of lost knowledge from the dataset) [4, 14, 39]. In the context of DIACHRON, the FOL queries that are described above, are posed as SPARQL queries over a dataset instance that lies in the triple store. This procedure is described in 4.1.5 below.

### 4.1.4 Overview of the Validation and Repairing Service Architecture

The Validation and Repairing Service will be based on the diagnosis and repair module, which consists of three main components: i) the input component, ii) the diagnosis component and iii) the repair component. An overview of the module architecture and the respective workflow is given in Figure 23.

The module takes as input an OWL ontology (TBox) together with the information needed to connect to a named graph that resides in a triple store (ABox). The TBox represents the constraints that should be satisfied by the ABox. The input component is responsible for parsing the OWL ontology and storing it in main memory as well as for connecting to the triple store using the provided credentials and connection information.

![Figure 23: Diagnosis and Repair Module Architecture](image)

The main memory ontology model and the connection to the triple store are then given as input to the diagnosis component. The first step performed by the diagnosis component is to compute all the inferred constraints, called the closure of negative inclusions of the ontology (see Section 4.1.5 for details). This computation takes into
account only the TBox assertions that comply with the $DL-Lite_A$ specification. All the other assertions that may exist in the TBox are ignored.

The next step of the diagnosis component is to transform the constraints that appear in the closure of negative inclusions to their respective SPARQL queries (see 4.1.5) and then pose these queries over the named graph of the triple store. The answers to these queries form the set of invalidities that are diagnosed in the dataset. The diagnosis component is also responsible for representing these invalidities as an interdependency graph (see Subsection 4.1.5 for details on the interdependency graph), which is the main input of the repair component.

Finally, the repair component is responsible for performing the repair of the dataset, based on the diagnosis results. More specifically, the repair component automatically computes a delta which leads to a consistent dataset, and then applies this delta on the named graph that contains the dataset. The repair component is further explained in Subsection 4.1.6.

### 4.1.5 Diagnosis Component Details

The diagnosis component is used to determine and provide as output all the data assertions (ABox assertions) that are inconsistent with respect to one or more integrity constraints expressed in the underlying schema (TBox).

Before explaining the process of diagnosis, we have to make a distinction between the different $DL-Lite_A$ TBox assertions, namely between positive inclusions and negative inclusions. In a simplified - but adequate - manner, positive TBox inclusions are of the form $B_1 \sqsubseteq B_2$ (where $B_1$ and $B_2$ are concepts) or of the form $Q_1 \sqsubseteq Q_2$ (where $Q_1$ and $Q_2$ are roles). Respectively, negative TBox inclusions are of the form $B_1 \sqsubseteq \neg B_2$ or of the form $Q_1 \sqsubseteq \neg Q_2$. This distinction is important due to the fact that the ABox is viewed under the Open World Assumption (OWA). Due to the OWA, a positive inclusion can never be violated; therefore, the only interesting (from the diagnosis perspective) constraints are the negative inclusions. Therefore, the notions of negative TBox inclusions and constraints will be used as identical.

Despite the above fact, positive inclusions are also relevant for the diagnosis process, because they may generate inferred information that should be taken into account. To see this, assume that the TBox contains the constraint $A \sqsubseteq \neg C$ and the assertion $B \sqsubseteq C$, and suppose that the dataset contains both $A(x)$ and $B(x)$ for some $x$. Even though no constraint is explicitly violated, the combination of the dataset contents with the aforementioned TBox would lead to inferring both $C(x)$ and $\neg C(x)$, i.e., an invalidity. Note that the positive inclusion $B \sqsubseteq C$, albeit not violated, plays a critical role in creating this invalidity.

Capturing such implicit invalidities can be done by computing the closure of the dataset, which will infer the fact that both $C(x)$ and $\neg C(x)$ are implied by the dataset (as a consequence of the explicit ABox and TBox assertions). Unfortunately, computing such a closure for large datasets is not efficient.

To avoid this efficiency problem, the diagnosis algorithm uses a technique which identifies the constraints that are implied by the explicitly declared constraints and assertions in the schema. In our example, we could identify that the constraint $A \sqsubseteq \neg B$ is a consequence of the two constraints in the ABox, so the presence of $A(x)$ and $B(x)$ violates this implicit constraint. This process amounts to computing the closure of negative inclusions and functionality assertions of the TBox [13], i.e., the set of all the explicit and implicit functionality assertions and negative inclusions present in the TBox. This method is generally more efficient, as the number of constraints is usually much smaller than the size of the dataset.

In addition, it has been proven that, in order to check the satisfiability of a $DL-Lite_A$ KB (therefore diagnose the KB for invalidities), one has to take into account only the closure of negative inclusions and functionality assertions. More precisely, it is shown that an invalidity in a $DL-Lite_A$ KB exists only if an ABox assertion contradicts a functionality assertion or a negative inclusion included in the closure of negative inclusions and functionality assertions [11].

Thus, the first step of the diagnosis algorithm is the computation of the closure of negative inclusions and functionality assertions (cf. Figure 23), in order to get the full set of constraints (expressed in $DL-Lite_A$ syntax) that need to be checked over the database. This computation is made by closing the set of negative inclusions and functionality assertions, taking into account the interaction between positive and negative inclusions. To perform this, we use the algorithm described in [11].

Having computed the closure of negative inclusions and functionality assertions, the resulting constraints are
transformed to SPARQL queries using predefined patterns, which are then posed on the database. This transformation is defined as follows, depending on the type of the constraint \( T \) that appears in the closure of negative inclusions and functionality assertions:

- **Case** \( T = X_1 \sqsubseteq \neg X_2 \) and it is a concept inclusion:
  - If \( X_i \) is in the form of a concept \( A \), the query should check if \( A(x) \) appears in the database.
  - If \( X_i \) is in the form of \( \exists P \), the query should check if \( P(x, u) \) appears in the database.
  - If \( X_i \) is in the form of \( \exists P^- \), the query should check if \( P(u, x) \) appears in the database.
  - If \( X_i \) is in the form of \( \delta(U_C) \), the query should check if \( U_C(x, u) \) appears in the database.
  - If \( X_i \) is in the form of \( \exists \delta(U_R) \), the query should check if \( U_R(x, u, w) \) appears in the database.

- **Case** \( T = X_1 \sqsubseteq \neg X_2 \) and it is a value-domain inclusion:
  - If \( X_i \) is in the form of a value-domain \( D \), the query should check if \( D(x) \) appears in the database.
  - If \( X_i \) is in the form of \( \rho(U_C) \), the query should check if \( U_C(u, x) \) appears in the database.
  - If \( X_i \) is in the form of \( \rho(U_R) \), the query should check if \( U_R(u, w, x) \) appears in the database.

- **Case** \( T = X_1 \sqsubseteq \neg X_2 \) and it is a role inclusion:
  - If \( X_i \) is in the form of a role \( P \), the query should check if \( P(x, y) \) appears in the database.
  - If \( X_i \) is in the form of \( P^- \), the query should check if \( P(y, x) \) appears in the database.
  - If \( X_i \) is in the form of \( \delta(U_R) \), the query should check if \( U_R(x, y, u) \) appears in the database.
  - If \( X_i \) is in the form of \( \delta(U_R)^- \), the query should check if \( U_R(y, x, u) \) appears in the database.

- **Case** \( T = X_1 \sqsubseteq \neg X_2 \) and it is a concept attribute inclusion:
  - If \( X_1 = U_{C_1} \) and \( X_2 = U_{C_2} \), the query should check if both \( U_{C_1}(x, y) \) and \( U_{C_2}(x, y) \) appear in the database.

- **Case** \( T = X_1 \sqsubseteq \neg X_2 \) and it is a role attribute inclusion:
  - If \( X_1 = U_{R_1} \) and \( X_2 = U_{R_2} \), the query should check if both \( U_{R_1}(x, y, z) \) and \( U_{R_2}(x, y, z) \) appear in the database.

- **Case** \( T = \text{(funct } X) \)
  - If \( X = P \), the query should check if \( P(x, y) \) and \( P(x, z) \) appear in the database, with \( y \neq z \).
  - If \( X = P^- \), the query should check if \( P(y, x) \) and \( P(z, x) \) appear in the database, with \( y \neq z \).
  - If \( X = U_C \), the query should check if \( U_C(x, y) \) and \( U_C(x, z) \) appear in the database, with \( y \neq z \).
  - If \( X = U_R \), the query should check if \( U_R(w, x, y) \) and \( U_R(w, x, z) \) appear in the database, with \( y \neq z \).

The transformation of the above FOL queries to their corresponding SPARQL queries is illustrated in the following example:

**Example 1.** Suppose that the closure of negative inclusions and functional assertions contains the following three constraints:

1. \( \text{(funct } p_1) \), where \( p_1 \) is a property (role)
2. \( c_1 \sqsubseteq \neg c_2 \), and both \( c_1 \) and \( c_2 \) are classes (concepts)
3. \( \exists p_1 \sqsubseteq \neg c_1 \), where \( p_1 \) is a property (role) and \( c_1 \) is a class (concept)
Then, the corresponding SPARQL queries for the above constraints are the following:

1. SELECT ?x ?y1 ?y2
   WHERE {
     ?x p1 ?y1.
     ?x p1 ?y2.
     FILTER (?y1 != ?y2)
   }
2. SELECT ?x
   WHERE {
     ?x rdf:type c1.
     ?x rdf:type c2
   }
3. SELECT ?x
   WHERE {
     ?x p1 ?y.
     ?x rdf:type c1
   }

The answers to those queries describe the invalidities that are diagnosed in the database.

It should be noted here that, due to the DL-LiteA language specification, only unary expressions can appear in either side of a TBox inclusion assertion. That leads to the conclusion that an invalidity in the database is always caused by pairs of data assertions. This makes it easier to compute possible repairs, in the sense that the repair of an invalidity is always possible by removing either one of the two assertions taking part in this invalidity.

The last step of the diagnosis process is to encode the invalidities found by the execution of the aforementioned queries in a convenient form, namely in the form of an interdependency graph. The interdependency graph encodes in a compact way all the invalidities that were found during the diagnosis process, as well as the interdependencies between data assertions (related to such invalidities). An interdependency between two data assertions exists if there is an invalidity caused by the existence of this pair of assertions in the database, which can be resolved by the removal of either one of these assertions. This is described in the following example:

**Example 2.** Suppose that the TBox $T$ contains the following two constraints:

$$c1 \sqsubseteq \neg c2$$

(funct $p1$)

Also suppose that the ABox $A$ contains the following assertions:

$$c1(x1)$$
$$c2(x1)$$
$$p1(x2,x3)$$
$$p1(x2,x4)$$
$$p1(x2,x5)$$

Then, after checking the constraints of $T$ over $A$, the following pairs of data assertions cause invalidities (each assertion of a pair is interdependent with the other assertion of the pair):

$$[c1(x1), c2(x1)]$$
$$[p1(x2,x3), p1(x2,x4)]$$
$$[p1(x2,x3), p1(x2,x5)]$$
$$[p1(x2,x4), p1(x2,x5)]$$

These interdependencies can be neatly represented in a graph structure (the interdependency graph), which is then provided as input to the repair algorithm. The interdependency graph consists of invalid data assertions, as vertices, and interdependencies between invalid data assertions, as edges (see Figure 24). It is produced by iterating over all pairs of data assertions that cause invalidities (provided by the previous step of the algorithm) and
by adding these assertions to the graph, as vertices. For each pair of vertices in the graph that represents a pair of interdependent data assertions, an edge connecting these two vertices is added to the graph and labelled with the constraint instance that this pair of assertions break.

Note that the graph does not contain duplicate vertices, meaning that, a data assertion will appear at most once in the graph, regardless of how many invalidities it is involved in. Note however that, for each invalidity that said assertion is involved in, a different edge connecting this vertex (data assertion) with the vertex that represents the other assertion of the pair in the invalidity, is added to the graph. As a result, we can easily determine how many invalidities each data assertion is involved in. This information will be used during repair (see Subsection 4.1.6).

Figure 24 helps in illustrating the structure of the interdependency graph and the way it is constructed. This figure presents the interdependency graph produced by example 2, expressed in OWL syntax.

![Figure 24: Example of an interdependency graph](image)

The use of the interdependency graph as a structure to represent the invalidities that are diagnosed in the database, gives the ability to get a better grasp of the form and the complexity of interdependencies, to visualize these interdependencies in a way that is easily understood (using graph visualization tools), and to use methods and tools that come from graph theory in order to facilitate the repair process (see Subsection 4.1.6 for more details).

### 4.1.6 Repair Component Details

The repair component is responsible for automatically computing and applying a delta that leads to a consistent dataset. It takes as input the interdependency graph which was previously constructed by the diagnosis component; this graph contains all the information needed to describe the diagnosed invalidities and to lead the repair algorithm.

As explained above, due to the form of \( DL-Lite_\alpha \) constraints, the repair of an invalidity is performed by the removal of either one of the two assertions that take part in this invalidity. Thus, in terms of the interdependency graph, resolving an invalidity amounts to selecting one of the two vertices that are connected by the edge representing this invalidity, and removing it. A full repair amounts to repeating this process for all edges in the graph.

This process is actually a direct match to the well-known problem of **minimal vertex cover**. A vertex cover of a graph is a set of vertices such that each edge of the graph is incident to at least one vertex of this set. By computing a vertex cover of the interdependency graph, the repair algorithm computes a delta that, when applied to the dataset, leads to a consistent dataset. This is due to the fact that the removal of all the vertices (in other
words, all the data assertions) in the vertex cover leads to the removal of all the edges (in other words, all the invalidities) from the graph. Note that we are interested in minimal, with respect to set inclusion, vertex covers, in order to guarantee minimal impact of the repairing process on the dataset (i.e., remove the minimal amount of assertions, cf. [4, 14, 39]).

We chose to compute the minimal vertex cover in a greedy manner, meaning that, in each step of the computation, the vertex that is chosen to be included in the cover is the vertex with the greatest degree (in other words, the data assertion that is part of the most interdependencies/violations). Note that this will not always lead to the optimal solution; however, the computation of the optimal solution is not efficient, as vertex cover is a known NP-complete problem. The simple approximation algorithm described above guarantees that the best choice is made locally (i.e., in each step of the algorithm) and is very efficient. It can be proven that the size of the vertex cover produced using the above approximation algorithm is at most $O(\log n)$ larger than the optimal one, where $n$ is the number of vertices of the interdependency graph.

The vertices included in the computed vertex cover represent the repair, i.e., the data assertions to be removed from the dataset in order to render it valid (called the *repairing delta*). Thus, the repairing algorithm needs next to apply the repairing delta to the dataset by removing those data assertions that belong to it. To remove those data assertions we execute a SPARQL Update statement, which can be easily constructed from the repairing delta and has the form illustrated in Example 3.

**Example 3.** Suppose that the vertex cover of the interdependency graph of Figure 24 contains the following three triples:

- $(x_1, \text{rdf}:\text{type}, c_1)$
- $(x_2, p_1, x_3)$
- $(x_2, p_1, x_4)$

The SPARQL Update query that repairs the database is the following:

```
DELETE DATA {
    x1 \text{rdf}:\text{type} c_1.
    x2 p_1 x_3.
    x2 p_1 x_4
}
```

Note that the repairing process described above is fully automatic (based on the vertex cover) and does not consider or require any user input. In addition to this automatic repairing approach, we are also considering an entirely manual approach (where the user will be asked to resolve all invalidities by himself, based on the interdependency graph, possibly with the help of some intuitive navigation interface), as well as a semi-automatic approach (which will be based on user preferences over the possible invalidities and their possible resolutions, in a manner similar to [39], and will possibly employ a minimal amount of interaction with the user as well).

### 4.1.7 Repairing in DIACHRON

The above discussion is related to a generic approach for handling diagnosis and repair in RDF graphs, where the constraints are expressed in DL-Lite$_A$. In the context of this project, the described techniques for diagnosis and repair of a database have to be aligned with the specific DIACHRON data model. This can be done using one of the two approaches described below.

Under the first approach, we express the constraints directly in the DIACHRON data model. In other words, we take each pilot’s constraints (which can be expressed at any language) and identify what they correspond to in the representation of the pilot’s datasets in the DIACHRON data model.

This approach was rejected, because the constraints of the pilots, if expressed under the representation imposed by the DIACHRON data model, would generally be in a quite complex form that would fall outside of the class of constraints that can be captured by DL-Lite$_A$ axioms, leading to efficiency problems. This is true even if the original constraint was a simple one (e.g., disjointness).

The second approach uses the original constraints of the pilots to perform diagnosis and repairing, rather than their translation in the DIACHRON data model. In other words, the constraints whose validity the repairing service will check are expressed in a form that is close to the pilot’s original language, and the repairing delta is also expressed in the pilot’s representation.
To apply this idea over the DIACHRON system, where the datasets will be actually stored in the DIACHRON data model, we will need a wrapper that acts as a mediator between the repairing service and the DIACHRON data model. This wrapper manages the data accesses of the repairing service in a way that does not require it to translate the various requests into the DIACHRON data model. In other words, the repairing service is interacting with the data in a way that is convenient for the repairing service, and this interaction is then “translated” into adequate interaction in the underlying model.

To be more specific, let’s consider any given pilot. The first step will be to take each constraint of the pilot and determine what type of $\text{DL-Lite}_A$ axiom it corresponds to (see Appendix C for a complete description of the types of constraints that $\text{DL-Lite}_A$ can handle). As an example, a primary key constraint in the relational model would correspond to a functional property in a straightforward representation of the relational model to RDF. During diagnosis, the repairing service will generate a set of SPARQL queries to be executed to determine the invalidities. As the constraints are expressed in terms of the pilot’s native format, the SPARQL queries generated will also be expressed in those terms; at this point, the wrapper will take care of “translating” these SPARQL queries to their corresponding SPARQL queries under the DIACHRON model representation (see also Example 4).

After the invalidities have been returned and the repairing delta has been computed, the repairing service will return the SPARQL Update statement that determines what needs to be deleted; this SPARQL Update will, again, be written using the pilot’s native terminology, so the wrapper will again take care of the translation to the DIACHRON model. Continuing the above example, the repairing of a primary key constraint would correspond to the deletion of one of the violating tuples, so this needs to be translated to the deletion of all the triples in the underlying data model that represent the tuple to be deleted.

In Appendix D, there is an illustration of the conversion of some logical scientific data constraints expressed in natural language to their respective $\text{DL-Lite}_A$ formulation. In the following, we present also an example of translating the SPARQL queries used for diagnosis into the required format to be applied in the DIACHRON model.

**Example 4.** Suppose that the closure of negative inclusions and functional assertions contains the following three constraints:

1. $(\text{funct } p_1)$, where $p_1$ is a property (role)
2. $c_1 \sqsubseteq \neg c_2$, and both $c_1$ and $c_2$ are classes (concepts)
3. $\exists p_1 \sqsubseteq \neg c_1$, where $p_1$ is a property (role) and $c_1$ is a class (concept)

Then, the corresponding SPARQL queries for the above constraints, after their translation in the DIACHRON model, are the following:

1. SELECT ?x ?y1 ?y2
   WHERE {
   ?rec1 rdf:type diachron:Record.
   ?rec1 diachron:subject ?x.
   ?rec1 diachron:hasAttr ?attr1.
   ?attr1 diachron:hasAttr ?attr1.
   ?rec2 rdf:type diachron:Record.
   ?rec2 diachron:subject ?x.
   ?rec2 diachron:hasAttr ?attr2.
   FILTER(?y1 != ?y2)}

2. SELECT ?x
   WHERE {
   ?rec1 rdf:type diachron:Record.
4.2 Cleaning

4.2.1 Motivation

A crucial prerequisite for realizing the Semantic Web vision is the availability of high quality data in a machine understandable form. In line with Deliverable 5.1 [46] “quality assessment and quality-driven acquisition” we define quality as fitness for use; in particular, quality of semantic data refers to the following aspects: i) capturing the meaning of the situation they describe and ii) correct representation regarding the ontologies and vocabularies employed. Steps in the life cycle of semantic data include creation, conversion, enrichment, publishing and interlinking. Each of these steps is assigned with data transformations that can potentially have a lasting effect on the quality. Numerous problems happen because of, e.g., misspelling, missing values, literals malformed w.r.t. their data types, or erroneous statements.

Deliverable 5.1 [46] presented a detailed overview of common data quality problems and defined how to measure and quantify them using quality metrics. However, not all quality problems considered in this deliverable could be addressed by cleaning. Those problems that are caused by violation of a purely logical constrain are issues of repairing and are described in the previous section 4.1. Cleaning refers to detection of incomplete, incorrect, inaccurate, irrelevant, etc. facts in the data and then replacing, modifying, or deleting them. In different literature sources cleaning is also called as data cleansing or scrubbing. There is no commonly agreed upon definition of this term, because it strongly depends on particular application domains. Data quality and data cleaning are an area of interest in research as well as in business communities. Cleaning is an important issue in data warehousing, knowledge discovery in data bases and in data/information management. In [21] cleaning is defined as the process of eliminating errors and inconsistencies in data and solving the object identity problem. [25] defines the data cleansing problem as the merge/purge problem. In data warehousing data cleansing is the process of analyzing the quality of data in a data source, manually approving/rejecting the suggestions by the system, and thereby making changes to the data [2].

The best strategy for addressing quality issues is to understand their root causes:

**Typographical Errors and Non-Conforming Data** Despite automation, data is still often manually filled into Web forms or other kinds of user interfaces. The most common cause for invalid values is the human factor
– the user manually entering the data just makes a mistake. He chooses the wrong entry from a list or enters a correct data value into the wrong field.

**Incomplete information** Data entered into a form might not be complete by mistake. Often data is not captured or people give incomplete or incorrect information for different reasons, e.g. to safeguard their privacy. If one field is not available, an alternate field is often used. This can lead to data quality issues such as misfielded data, wrong data types or unrealistic values. Having country names in the "city" field is one example for this problem.

**Data integration** Semantic data sets are rarely created from scratch. The starting point is typically a data conversion or/and integration from some previously existing data sources. When the original sources differ in format or/and used vocabularies, merging them increases the likelihood of data quality errors.

**Inconsistent Naming Conventions** Using different entry formats or abbreviations leads to incomplete or inaccurate values (e.g. ‘ONE’ instead of ‘1’).

Such problems significantly decrease the quality of data and thus restrict their fitness for use. Moreover, incomplete, incorrect, inaccurate, irrelevant literals or URI’s reduce therefore the number of properties known about an entity and hence lead to an inadequate representation of an entity.

These issues are addressed by the DIACHRON cleaning module, which aims to identify and then to improve quality problems related to the task of cleaning. Cleaning should meet the following requirements:

1. Measuring the quality of a data set is required to decide on the necessity of cleaning, and also to measure the success of cleaning afterwards.

2. Supporting detection and distinction of quality problems. This is to ensure that all quality problems relevant to cleaning are detected.

3. Supporting the identification of error instances in order to correct/remove them. This imply making use of the Diachron data model in order to easily handle (find, remove, modify) the erroneous instances.

4. Possibility to extend the existing framework by e.g. defining additional quality checks.

The DIACHRON cleaning module addresses all of these requirements by providing:

- A generic framework for determining quality status, classification of detected quality problems and for identifying invalid instances.

- A set of rules for detecting invalid or unrealistic values violating integrity constrains.

- A set of methods that define how to clean data regarding the identified problems.

Since cleaning targets errors in the data, its definition strongly depends on the definition of what these data errors are. In the next sections we first present those quality problems that can be addressed by cleaning. We then present a generic architecture of the cleaning module and describe its main component in more detail. Finally we give an overview of the cleaning methods used in DIACHRON.

### 4.2.2 Classification of Cleaning Problems

In order to provide the user with a precise and detailed report about quality problem we adopt the classification of values used relational databases, namely in the Microsoft SQL Server relational database management system [1]. It distinguishes between ‘correct’, ‘error’ and ‘invalid’ values, defined as follows:

- A value is **correct** if it i) has a correct data type (as specified by the corresponding schema), ii) belongs to the value domain, and iii) does not contain any syntax errors. Range is defined as a sequence of symbols from a finite, non-empty alphabet. For each domain there exists an arbitrary grammar describing the syntax of its values [33].
Table 5: Examples of Errors Related to the Constrains on Literals

<table>
<thead>
<tr>
<th>Group</th>
<th>Problem</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Syntactical</td>
<td>Domain format error</td>
<td>date ‘35.01.1978’</td>
<td>value outside of range</td>
</tr>
<tr>
<td></td>
<td>Abbreviations</td>
<td>sex ‘f’ instead of ‘female’</td>
<td>inconsistent naming convention</td>
</tr>
<tr>
<td></td>
<td>Misspellings</td>
<td>city ‘Aten’</td>
<td>typos, phonetic errors</td>
</tr>
<tr>
<td></td>
<td>Misfielded value</td>
<td>country= ‘Bonn’</td>
<td></td>
</tr>
<tr>
<td>Semantical</td>
<td>Contradictions</td>
<td>country= ‘Germany’,</td>
<td>violation of value dependencies</td>
</tr>
<tr>
<td></td>
<td></td>
<td>City= ‘Chicago’</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Malformed literals</td>
<td>weight='twenty''xsd:double</td>
<td>wrong data type</td>
</tr>
<tr>
<td></td>
<td>Duplicated records</td>
<td>name= ‘Paulo Coelho’</td>
<td>same person is represented twice</td>
</tr>
<tr>
<td></td>
<td></td>
<td>and name= ‘P. Coelho’</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Integrity constrains</td>
<td>weight= ‘-5’</td>
<td>violation of common sense</td>
</tr>
<tr>
<td>Coverage</td>
<td>Missing Values</td>
<td>phone= ‘9999–999999’</td>
<td>Not captured data</td>
</tr>
</tbody>
</table>

- A value has an **error** if it belongs to the range, but is an incorrect value. For example, `name.googlemail.com` instead of `name@googlemail.com` in an email domain is an error. Although it is not correct email string it could be easily corrected to a valid value.

- A value is **invalid** if it does not belong to the domain, and does not have a correction. For example, the value “Germany” in the email domain is invalid, since it can not be corrected to the valid value.

A rough classification of quality problems comprises the following three groups: syntactical, semantical and coverage problems.

- **Syntactical** errors are violations regarding the data format. They include, e.g., misplaced errors (a correct value is entered into the wrong field), range format errors (values not in the domain format) and irregularities (non-uniform use of values)

- **Semantical** errors are violations of the data model. This includes, e.g., contradictions – values in one or several triples that violate some dependencies between the value predicates. Another example for this group are duplicates – two entities in the data set representing the same real world object. Since in the most cases duplicate entities slightly differ in their property values (e.g. person name ‘Peter Schmidt’ vs ‘P. Schmidt’ it is a challenge to find them out.

- **Coverage** errors are missing values.

Table 5 lists some examples for the errors described above [38].

Data quality has multiple dimensions, which can be grouped into a set of categories, such as accessibility, trust or intrinsic dimensions. Deliverable 5.1 presents a comprehensive survey on existing methodologies for assessing data quality, resulting in a large set of quality metrics. Table 6 lists a selection of those relevant for cleaning.

### 4.2.3 Cleaning Module Architecture

The data cleaning process is defined as a set of operations on data in order to remove, correct or modify data affected by quality problems as described in 4.2.2.

The Diachron data cleaning module comprises the following components:
1. Quality status assessment.

2. Knowledge base, containing i) common data schemas, ii) a set schema/data set specific constraints on literals, iii) a set of schema/data set cleaning rules that could be extended by the users.

3. Generation of cleaning suggestions.

4. User interface for interactively review and approval of cleaning suggestions.

Figure 25 shows the architecture of the Diachron cleaning module.

![Diachron Cleaning Module Architecture](image)

Figure 25: Architecture of DIACHRON Cleaning Module

Data cleaning is viewed as a process which begins with a data source the user wants to improve/check regarding its quality. Once the user starts the cleaning process, the quality assessment component loads from the knowledge base the data schemas used in data set. The core of the quality status assessment component is a set of quality metrics developed and implemented in the context of WP5, particularly in the Task 5.1 ‘Quality Metrics for LOD’ and the Task 5.2 ‘Ranking LOD Datasets for Appraisal’. Using the quality metrics data is audited to detect inconsistencies, and a quality status report is generated. Support for different levels of result reporting is provided according to the user’s needs and to his knowledge. The conceptual structure of the quality report is presented by Figure 26.

The quality report can be used for both summarization of a quality status and detailed representation of occurring quality problems. The Box A in the Figure 26 shows the conceptual model of the quality report, the Box B extends the previous figure by the model of quality metric. The quality metrics data structures shown in Box B are intended to be stored persistently to aid filtering and ranking datasets by quality; therefore, the data model in Box B has been implemented as an RDF vocabulary (cf. Task 5.2). In contrast, the quality report data structures shown in Box A need not be stored persistently, as we intend a quality report to be reviewed by the maintainers of a dataset, who will respond to it by fixing quality issues.

The main concept is a ‘Quality report’, which is ‘computesOn’ a ‘Data set’ and ‘contains’ a set of ‘Quality problems’. A ‘Quality problem’ is again ‘describedBy’ by the corresponding ‘Quality metric’ and ‘affects’ a set of ‘Entities’. The Box A represents the structure of ‘Quality metric’, which ‘has’ a ‘Name’ and belongs to the certain ‘Dimension’. This model for the quality report serves a basis to specify a sequence of operation required for cleaning, so called ‘cleaning suggestions’ which include:
Derivation of missing values from existing ones. Although there is no general approach how to do it automatically for all data sets, a set of schema/data set specific rules created by the user and stored in the knowledge base is a feasible solution for this problem.

- Removing contradictions within an entity
- Merging or eliminating duplicates
- Removing properties containing values that violate constraints on literals (wrong data type, out of range).

The generated cleaning suggestions are then shown to the user for reviewing and approval. In the next step the approved cleaning operation are applied to the data set, and finally a report about the performed changes is generated.
Table 6: Quality Problems

<table>
<thead>
<tr>
<th>Dimension</th>
<th>Quality Metric</th>
<th>Description</th>
<th>Cleaning approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Availability</td>
<td>SPARQL endpoint accessibility</td>
<td>indicates whether data set can be accessed through SPARQL endpoint</td>
<td>Problem report</td>
</tr>
<tr>
<td></td>
<td>RDF dump accessibility</td>
<td>indicates whether a RDF dump is provided and can be downloaded</td>
<td>Problem report</td>
</tr>
<tr>
<td></td>
<td>dereferencability of links</td>
<td>Detection of not valid URI’s or broken links</td>
<td>Problem report</td>
</tr>
<tr>
<td></td>
<td>availability of structured data</td>
<td>detection of dead links or detection of a URI without any supporting</td>
<td>Problem report</td>
</tr>
<tr>
<td></td>
<td>dereferencability of back-links</td>
<td>detection of no dereferenced back-links</td>
<td>Problem report</td>
</tr>
<tr>
<td></td>
<td>dereferencability of forward-links</td>
<td>detection of no dereferenced forward-links</td>
<td>Problem report</td>
</tr>
<tr>
<td>Accuracy</td>
<td>Inaccurate values</td>
<td>detection of values violating dependencies rules</td>
<td>Rule-based value modification, Problem report</td>
</tr>
<tr>
<td></td>
<td>malformed datatype literals</td>
<td>detection of wrong typed literals regarding lexical syntax</td>
<td>Autom. modification, rule-based value modifi-</td>
</tr>
<tr>
<td></td>
<td>literals incompatible with datatype range</td>
<td>detection of wrong typed literals</td>
<td>cation, problem report</td>
</tr>
<tr>
<td></td>
<td>erroneous annotation</td>
<td>detection of error instances</td>
<td>Problem report</td>
</tr>
<tr>
<td></td>
<td>inaccurate annotation, labelling, classification</td>
<td>detection of inaccurately annotated instances</td>
<td>Problem report</td>
</tr>
<tr>
<td>Conciseness</td>
<td>duplicate instance</td>
<td>detection of instances representing the same real world object</td>
<td>merging of duplicates</td>
</tr>
<tr>
<td>Representation</td>
<td>keeping URIs short</td>
<td>detection of long URIs or those that contain query parameters</td>
<td>rule based modification</td>
</tr>
<tr>
<td>Licensing</td>
<td>permission to use data set</td>
<td>detection of license for reproduction, distribution, modification or redistribution</td>
<td>Automatic modification</td>
</tr>
</tbody>
</table>

Grant Agreement No. 601043
5 Conclusion

WP3 has a central position in the DIACHRON vision, as it deals with the critical problems of identifying, recording, monitoring and propagating changes between versions of DIACHRON entities (such as datasets), as well as with restoring certain aspects of data quality (namely, validity, consistency and correctness), which are often jeopardized due to the open evolution of remote datasets. This deliverable describes the progress made in various problems related to WP3, namely the definition of changes, representation of changes, detection of changes, change monitoring and propagation, repairing, and cleaning.

For each of the above problems, we define the scope and context in which they are considered, motivate the need for addressing them in the context of the DIACHRON vision and explain the approach used for solving them. The upcoming first deployment of the DIACHRON platform (to appear in the prototype deliverable D3.2, due on M16 of the project) will build on the above developments in order to implement the WP3 services, namely change detection, change monitoring and propagation, repairing, and cleaning.
References


A Basic Changes in the DIACHRON Language of Changes

In this appendix, we list the basic changes that will be used in DIACHRON. As mentioned in the main text, we are reusing the basic changes described in [34]. Note that these changes are generic and not specific to the DIACHRON model, and they apply equally to any RDF dataset.

In the tables below we list, for each change:

- Its name.
- The intuition it captures, described in terms of the RDF model.
- Its parameters, and the intuition behind each parameter.
- The SPARQL query which will be used for its detection (cf. Section 2.3).

### Basic Changes in the DIACHRON Language of Changes

<table>
<thead>
<tr>
<th>Change</th>
<th>Add_Type_Class(a)</th>
<th>Delete_Type_Class(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add object a of type rdfs: class</td>
<td>Delete object a of type rdfs: class</td>
</tr>
<tr>
<td>Parameters</td>
<td>a = The added object</td>
<td>a = The deleted object</td>
</tr>
<tr>
<td>δ⁺</td>
<td>(a, rdf : type, rdfs : class).</td>
<td>a does not appear in V₁</td>
</tr>
<tr>
<td>δ⁻</td>
<td>(a, rdfs : subClassOf, rdfs : resource)</td>
<td>a does not appear in V₂</td>
</tr>
<tr>
<td>φ</td>
<td>a does not appear in V₁</td>
<td>a does not appear in V₂</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Change</th>
<th>Add_Type_Property(a)</th>
<th>Delete_Type_Property(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add object a of type rdf : property</td>
<td>Delete object a of type rdf : property</td>
</tr>
<tr>
<td>Parameters</td>
<td>a = The added object</td>
<td>a = The deleted object</td>
</tr>
<tr>
<td>SPARQL used for detection</td>
<td>SELECT ?s WHERE { GRAPH &lt;added&gt; {?a rdf:type rdf:Property.} FILTER NOT EXISTS { GRAPH &lt;V1&gt; {{?a ?p ?o} UNION {?s ?p2 ?a } UNION {?s1 ?a ?o1 }} } }</td>
<td>SELECT ?s WHERE { GRAPH &lt;deleted&gt; {?a rdf:type rdf:Property.} FILTER NOT EXISTS { GRAPH &lt;V2&gt; {{?a ?p ?o} UNION {?s ?p2 ?a } UNION {?s1 ?a ?o1 }} } }</td>
</tr>
<tr>
<td>δ⁺</td>
<td>(a, rdf : type, rdf : property)</td>
<td>φ</td>
</tr>
<tr>
<td>δ⁻</td>
<td>φ</td>
<td>a does not appear in V₂</td>
</tr>
<tr>
<td>φ</td>
<td>a does not appear in V₁</td>
<td>a does not appear in V₂</td>
</tr>
</tbody>
</table>

The changes Add_Type_Individual and Delete_Type_Individual are defined analogously with the exception that (a, rdf : type, rdfs : resource) should be in δ⁺ (δ⁻) instead of (a, rdf : type, rdf : property).
Grant Agreement No. 601043

<table>
<thead>
<tr>
<th>Change</th>
<th>Retype Class To Property(a)</th>
<th>Retype Property To Class(a)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Retype class a to a property</td>
<td>Retype property a to a class</td>
</tr>
<tr>
<td>Parameters</td>
<td>a = The retyped object</td>
<td>a = The retyped object</td>
</tr>
</tbody>
</table>

**SPARQL used for detection**

For detection:

- **Retype Class To Property**

- **Retype Property To Class**

**Parameters**

- `a` = The retyped object

**Intuition**

- Retype class to a property
- Retype property to a class

**The rest of the retyping operations, namely:**

- **Retype Class To Individual(a)**
- **Retype Class To Property(a)**
- **Retype Individual To Class(a)**
- **Retype Individual To Individual(a)**
- **Retype Property To Class(a)**
- **Retype Property To Individual(a)**

are defined analogously (details omitted).

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Superclass(a,b)</th>
<th>Delete Superclass(a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Parent b of class a is added</td>
<td>Parent b of class a is deleted</td>
</tr>
<tr>
<td>Parameters</td>
<td>a = The class</td>
<td>a = The class</td>
</tr>
<tr>
<td>b = The new parent</td>
<td>b = The old parent</td>
<td></td>
</tr>
</tbody>
</table>

**SPARQL used for detection**

- **Add Superclass**

- **Delete Superclass**

**Parameters**

- `a` = The class
- `b` = The new parent

**Intuition**

- Parent b of class a is added
- Parent b of class a is deleted

**SPARQL used for detection**

- **Add Superproperty**
  - `SELECT ?a, ?b WHERE { GRAPH <added> { ?a rdfs:subPropertyOf ?b. } }`

- **Delete Superproperty**
  - `SELECT ?a, ?b WHERE { GRAPH <deleted> { ?a rdfs:subPropertyOf ?b. } }`

**Parameters**

- `a` = The property
- `b` = The new parent

**Intuition**

- Parent b of property a is added
- Parent b of property a is deleted

**SPARQL used for detection**

- **Add Superproperty**
  - `SELECT ?a, ?b WHERE { GRAPH <added> { ?a rdfs:subPropertyOf ?b. } }`

- **Delete Superproperty**
  - `SELECT ?a, ?b WHERE { GRAPH <deleted> { ?a rdfs:subPropertyOf ?b. } }`
<table>
<thead>
<tr>
<th>Change</th>
<th>Add Type To Individual(a,b)</th>
<th>Delete Type From Individual(a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Type b of individual a is added</td>
<td>Type b of individual a is deleted</td>
</tr>
<tr>
<td>Parameters</td>
<td>a = The individual</td>
<td>a = The individual</td>
</tr>
<tr>
<td></td>
<td>b = The new type (class)</td>
<td>b = The old type (class)</td>
</tr>
<tr>
<td>SPARQL used</td>
<td>GRAPH &lt;added&gt; {?a rdf:type ?b.}</td>
<td>GRAPH &lt;deleted&gt; {?a rdf:type ?b.}</td>
</tr>
<tr>
<td>for detection</td>
<td>FILTER (?b != rdfs:Resource).}</td>
<td>FILTER (?b != rdfs:Resource).}</td>
</tr>
<tr>
<td></td>
<td>GRAPH &lt;V2&gt; {(?a rdf:type rdfs:Resource.)}</td>
<td>GRAPH &lt;V1&gt; {(?a rdf:type rdfs:Resource.)}</td>
</tr>
</tbody>
</table>

| δ⁺   | (a, rdfs : type, b)                                              | ∅                                                                   |
| δ⁻   | ∅                                                                | (a, rdfs : type, b)                                                 |
| Φ    | a is an individual in V₂ ∧ b ≠ rdfs : resource                  | a is an individual in V₁ ∧ b ≠ rdfs : resource                      |

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Property Instance(a₁,a₂,b)</th>
<th>Delete Property Instance(a₁,a₂,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add property instance of property b</td>
<td>Delete property instance of property b</td>
</tr>
<tr>
<td>Parameters</td>
<td>a₁ = The subject</td>
<td>a₁ = The subject</td>
</tr>
<tr>
<td></td>
<td>a₂ = The object</td>
<td>a₂ = The object</td>
</tr>
<tr>
<td></td>
<td>b = The property</td>
<td>b = The property</td>
</tr>
<tr>
<td>SPARQL used</td>
<td>SELECT ?a₁, ?a₂, ?b WHERE {</td>
<td>SELECT ?a₁, ?a₂, ?b WHERE {</td>
</tr>
<tr>
<td>for detection</td>
<td>GRAPH &lt;added&gt; {?a₁ ?a₂ ?b.}</td>
<td>GRAPH &lt;deleted&gt; {?a₁ ?a₂ ?b.}</td>
</tr>
</tbody>
</table>

| δ⁺   | (a₁, b, a₂)                                                     | ∅                                                                  |
| δ⁻   | ∅                                                               | (a₁, b, a₂)                                                       |
| Φ    | b ≠ {rdfs : subClassOf, rdfs : subPropertyOf, rdfs : type,     | b ≠ {rdfs : subClassOf, rdfs : subPropertyOf, rdfs : type,         |
|      | rdfs : comment, rdfs : label, rdfs : domain, rdfs : range}    | rdfs : comment, rdfs : label, rdfs : domain, rdfs : range}        |

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Domain(a,b)</th>
<th>Delete Domain(a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Domain b of property a is added</td>
<td>Domain b of property a is deleted</td>
</tr>
<tr>
<td>Parameters</td>
<td>a = The property</td>
<td>a = The property</td>
</tr>
<tr>
<td></td>
<td>b = The domain</td>
<td>b = The domain</td>
</tr>
<tr>
<td>SPARQL used</td>
<td>SELECT ?a, ?b WHERE {</td>
<td>SELECT ?a, ?b WHERE {</td>
</tr>
<tr>
<td>for detection</td>
<td>GRAPH &lt;added&gt; {?a rdfs:domain ?b.}</td>
<td>GRAPH &lt;deleted&gt; {?a rdfs:domain ?b.}</td>
</tr>
</tbody>
</table>

| δ⁺   | (a, rdfs : domain, b)                                           | ∅                                                                 |
| δ⁻   | ∅                                                               | (a, rdfs : domain, b)                                            |
| Φ    |                                                                |                                                                  |

The changes Add Range and Delete Range are defined analogously with the exception that (a, rdfs : range, b) should be in δ⁺ (δ⁻) instead of (a, rdfs : domain, b).

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Comment(a,b)</th>
<th>Delete Comment(a,b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Comment b of object a is added</td>
<td>Comment b of object a is deleted</td>
</tr>
<tr>
<td>Parameters</td>
<td>a = The object</td>
<td>a = The object</td>
</tr>
<tr>
<td></td>
<td>b = The new comment</td>
<td>b = The old comment</td>
</tr>
<tr>
<td>SPARQL used</td>
<td>SELECT ?a, ?b WHERE {</td>
<td>SELECT ?a, ?b WHERE {</td>
</tr>
<tr>
<td>for detection</td>
<td>GRAPH &lt;added&gt; {?a rdfs:comment ?b.}</td>
<td>GRAPH &lt;deleted&gt; {?a rdfs:label ?b.}</td>
</tr>
</tbody>
</table>

<p>| δ⁺   | (a, rdfs : comment, b)                                         | ∅                                                                |
| δ⁻   | ∅                                                               | (a, rdfs : comment, b)                                          |
| Φ    |                                                                |                                                                  |</p>
<table>
<thead>
<tr>
<th>Change</th>
<th>\textit{Add Label}(a, b)</th>
<th>\textit{Delete Label}(a, b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Label (b) of object (a) is added</td>
<td>Label (b) of object (a) is deleted</td>
</tr>
<tr>
<td>Parameters</td>
<td>(a) = The object (b) = The new comment</td>
<td>(a) = The object (b) = The old comment</td>
</tr>
<tr>
<td>SPARQL used</td>
<td>\texttt{SELECT ?a, ?b WHERE { GRAPH &lt;added&gt; {?a rdfs:label ?b.} }}</td>
<td>\texttt{SELECT ?a, ?b WHERE { GRAPH &lt;deleted&gt; {?a rdfs:label ?b.} }}</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>((a, \texttt{rdfs:label}, b))</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\delta^-)</td>
<td>(\emptyset)</td>
<td>((a, \texttt{rdfs:label}, b))</td>
</tr>
<tr>
<td>(\phi)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>
B  Proposed Simple Changes for Pilots

In this appendix we present a list of simple changes to be considered in DIACHRON. They constitute our initial suggestions that may be modified by the corresponding pilots in the near future; thus, they should not be treated as the final list of simple changes to be considered in the project. We partitioned the appendix into three separate subsections, one per data model (relational, multidimensional, RDF). Note that the state of progress in each model is different, as explained in the corresponding subsections.

B.1  Relational Model

The proposed initial set of simple changes for the relational model (enterprise use case) are shown below, organized according to the type of entity they refer to. We consider three types of entities, namely table, column and tuple. As the list of changes has not been finalized yet, the formal definitions are still pending, so the tables below contain only the name, a short informal description of the change and information about the mapping to the DIACHRON model.

There are several issues that need to be resolved before finalizing the formal definitions, which are related mainly to the dependences between the different entities. For example, when deleting a table we should define if the corresponding deletion of associated columns and tuples should be reported as part of the Delete_Table, or as a separate change. Another similar example appears when merging tables. These issues will be examined closely with the help of the corresponding pilots in the upcoming months.
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Mapping to DIACHRON model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_Table</td>
<td>Create a new table as empty (no columns, no data defined).</td>
<td>A new instance of diachron:RelationalTable schema object is added in the new dataset version.</td>
</tr>
<tr>
<td>Delete_Table</td>
<td>Delete a table (having no effect on columns, or on data). Reverse of Add_Table.</td>
<td>The respective instance of diachron:RelationalTable schema object is deleted in the new dataset version.</td>
</tr>
<tr>
<td>Rename_Table</td>
<td>Rename the name of the table.</td>
<td>The neighborhood of the respective diachron:RelationalTable remains immutable. The mapping between uris is identified (add / delete the triple).</td>
</tr>
<tr>
<td>Merge_Table</td>
<td>Merge tables t1, t2, ..., tn into t. t contains the union of tuples in t1, t2, ..., tn.</td>
<td>A new instance of diachron:RelationalTable representing t is added. Those representing t1, t2, ..., tn are deleted. All instances of diachron:RelationalColumn in t1, t2, ..., tn are also deleted. For each column in t an instance of diachron:RelationalColumn is created and assigned to t (diachron:hasColumn), as well as its datatype (diachron:hasDataType).</td>
</tr>
<tr>
<td>Split_Table</td>
<td>Split table t into t1 given cond1, t2 given cond2, ..., tn given condn. Reverse of Merge_Table.</td>
<td>New instances of diachron:RelationalTable are created for each t1, t2, ..., tn. The instance of t is deleted. All instances of diachron:RelationalColumn in t are also deleted. For each column in t1, t2, ..., tn an instance of diachron:RelationalColumn is created, with a suitable datatype (diachron:hasDataType), and assigned (diachron:hasColumn).</td>
</tr>
</tbody>
</table>

Table 7: Proposed Simple Changes for the Relational Model (Entity: Table)
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Mapping to DIACHRON model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_Column</td>
<td>Add new column (affecting schema only, not data)</td>
<td>A new instance of diachron:RelationalColumn is added, assigned to the appropriate diachron:RelationalTable instance (diachron:hasColumn), with a given type (diachron:hasDataType).</td>
</tr>
<tr>
<td>Delete_Column</td>
<td>Delete column (affecting schema only, not data). Reverse of Add_Column.</td>
<td>The structure described in Add_Column is found deleted.</td>
</tr>
<tr>
<td>Modify_Column</td>
<td>Change the data type of column.</td>
<td>The respective triple with predicate diachron:hasDataType is deleted and another with a new datatype object is added.</td>
</tr>
<tr>
<td>Rename_Column</td>
<td>Change name of the column</td>
<td>The neighborhood of the diachron:RelationalColumn remains immutable.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>The mapping between uris is identified (add / delete the triple).</td>
</tr>
<tr>
<td>Copy_Column</td>
<td>Copy column c from table t1 to t2 with condition.</td>
<td>An instance of diachron:RelationalColumn is added for column c, assigned to the diachron:RelationalTable instance representing table t2, with an appropriate datatype it had in t1.</td>
</tr>
<tr>
<td>Move_Column</td>
<td>Copy column c from table t1 to t2 with condition, and then delete c from t1.</td>
<td>Same as copy and delete column c from table t1.</td>
</tr>
</tbody>
</table>

Table 8: Proposed Simple Changes for the Relational Model (Entity: Column)
### Entity: Tuple

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Mapping to DIACHRON model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_Tuple</td>
<td>Add a tuple.</td>
<td>A resource with type of the respective diachron:RelationalTable instance, is added.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>An instance of diachron:Record is added, grouping (via diachron:hasRecord:Attribute)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>all diachron:Record:Attribute instances interrelating a column instance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(diachron:predicate) and the respective value (diachron:object).</td>
</tr>
<tr>
<td></td>
<td></td>
<td>diachron:Record is connected with a diachron:subject property with the resource.</td>
</tr>
<tr>
<td>Delete_Tuple</td>
<td>Delete a tuple.</td>
<td>The structure described in Add_Tuple is deleted.</td>
</tr>
<tr>
<td>Update_Tuple</td>
<td>A single attribute value is updated.</td>
<td>The respective triple with predicate diachron:object is deleted and another with a new</td>
</tr>
<tr>
<td></td>
<td></td>
<td>value in object is added.</td>
</tr>
<tr>
<td>Add_Value_to_Tuple</td>
<td>Add a new value to a specific tuple.</td>
<td>A new diachron:Record:Attribute instance is added, interrelated with a column instance</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(diachron:predicate) and the respective value (diachron:object), and the diachron:Record</td>
</tr>
<tr>
<td></td>
<td></td>
<td>that the specific tuple corresponds to (diachron:hasRecord:Attribute).</td>
</tr>
<tr>
<td>Delete_Value_from_Tuple</td>
<td>The reverse operation of Add_Value_to_Tuple.</td>
<td>The structure described in Add_Value_to_Tuple is deleted.</td>
</tr>
<tr>
<td>Copy_Value_to_Tuple</td>
<td>Copy a value to a specific tuple. It is valid after a Copy_Column change.</td>
<td>Similar to Add_Value_to_Tuple. The added values should follow the condition in the</td>
</tr>
<tr>
<td>Move_Value_to_Tuple</td>
<td>Move a value from a specific tuple to another. It is valid after a Move_Column change.</td>
<td>it is same as Copy_Value_to_Tuple followed by a Delete_Value_from_Tuple.</td>
</tr>
</tbody>
</table>

Table 9: Proposed Simple Changes for the Relational Model (Entity: Tuple)
B.2 Multidimensional Model

The state of affairs for the multidimensional model (open data use case) is similar to that of the relational model. Below, the reader can find a proposed initial set of simple changes, organized according to the type of entity they refer to. In the multidimensional model we consider the following entities: measure, dimension, fact table, codelist/hierarchy (CH), attribute and observation.

Again, this list of changes is not finalized and the formal definitions are pending, so the tables below contain only the name, a short informal description of the change and information about the mapping to DIACHRON model. The finalization and formalization of the simple changes for the multidimensional model is expected to happen in the upcoming months with the help of the corresponding pilot.

<table>
<thead>
<tr>
<th>Entity: Measure</th>
<th>Description</th>
<th>Mapping to DIACHRON model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add Measure</td>
<td>Add new measure property (not assigned to a fact table, affects only schema)</td>
<td>Add a new instance of diachron:MeasureProperty, with a given datatype (diachron:hasDataType).</td>
</tr>
<tr>
<td>Delete Measure</td>
<td>Reverse of Add Measure.</td>
<td>The structure described in Add Measure is deleted.</td>
</tr>
<tr>
<td>Modify Measure_Type</td>
<td>Change the data type of measure.</td>
<td>The respective triple with predicate diachron:hasDataType is deleted and another with a new datatype object is added.</td>
</tr>
<tr>
<td>Rename Measure</td>
<td>Rename the measure.</td>
<td>The neighborhood of the diachron:MeasureProperty remains immutable. The mapping between uris is identified (add / delete the triple).</td>
</tr>
<tr>
<td>Attach Attr_to_Measure</td>
<td>Add an attribute property to measure.</td>
<td>A triple with diachron:hasAttribute predicate, and suitable measure as subject, and attribute as object is added.</td>
</tr>
<tr>
<td>Detach Attr_from_Measure</td>
<td>Delete an attribute property from measure. Reverse of Attach Attr_to_Measure.</td>
<td>The respective diachron:hasAttribute property is deleted.</td>
</tr>
</tbody>
</table>

Table 10: Proposed Simple Changes for the Multidimensional Model (Entity: Measure)
### Entity: Dimension

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Mapping to DIACHRON model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_Dimension</td>
<td>Add new dimension property (not assigned to a fact table, affects only schema).</td>
<td>Add a new instance of diachron:DimensionProperty, with a given datatype (diachron:hasDataType).</td>
</tr>
<tr>
<td>Delete_Dimension</td>
<td>Reverse of Add_Dimension.</td>
<td>The structure described in Add_Dimension is deleted.</td>
</tr>
<tr>
<td>Modify_Dimension_Type</td>
<td>Change data type.</td>
<td>The respective triple with predicate diachron:hasDataType is deleted and another with a new datatype object is added.</td>
</tr>
<tr>
<td>Rename_Dimension</td>
<td>Rename a dimension.</td>
<td>The neighborhood of the diachron:DimensionProperty remains immutable.</td>
</tr>
<tr>
<td>Attach_Attr_to_Dimension</td>
<td>Add an attribute property to dimension.</td>
<td>A triple with diachron:hasAttribute predicate, and suitable dimension as subject, and attribute as object is added.</td>
</tr>
<tr>
<td>Detach_Attr_from_Dimension</td>
<td>Delete an attribute property from dimension.</td>
<td>The respective diachron:hasAttribute property is deleted.</td>
</tr>
</tbody>
</table>

Table 11: Proposed Simple Changes for the Multidimensional Model (Entity: Dimension)

### Dimension: Fact Table

<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Mapping to DIACHRON model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_Fact_Table</td>
<td>Add new fact table (as empty table).</td>
<td>Add a new instance of diachron:FactTable.</td>
</tr>
<tr>
<td>Delete_Fact_Table</td>
<td>Reverse of Add_Fact_Table.</td>
<td>The structure defined in Add_Fact_Table is deleted.</td>
</tr>
<tr>
<td>Attach_Dimension_to_FT</td>
<td>Add a dimension property to fact table.</td>
<td>A diachron:hasDimension property is added, with suitable fact table as subject and dimension as object.</td>
</tr>
<tr>
<td>Attach_Measure_to_FT</td>
<td>Add a measure property to fact table.</td>
<td>A diachron:hasMeasure property is added, with suitable fact table as subject and measure as object.</td>
</tr>
<tr>
<td>Detach_Dimension_to_FT</td>
<td>Reverse of Attach_Dimension_to_FT.</td>
<td>The respective diachron:hasDimension property is deleted.</td>
</tr>
<tr>
<td>Detach_Measure_to_FT</td>
<td>Reverse of Attach_Measure_to_FT.</td>
<td>The respective diachron:hasMeasure property is deleted.</td>
</tr>
</tbody>
</table>

Table 12: Proposed Simple Changes for the Multidimensional Model (Entity: Fact Table)
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Mapping to DIARCHRON model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Delete_CH</td>
<td>Delete a codelist/hierarchy. Reverse of Add_CH.</td>
<td>An instance of diachron:CodeList / diachron:Hierarchy is deleted.</td>
</tr>
<tr>
<td>Attach_CH_to_Dimension</td>
<td>Assign a codelist/hierarchy to a dimension.</td>
<td>A diachron:CodeList property instance is added (connecting dimension and codelist/hierarchy instance).</td>
</tr>
<tr>
<td>Detach_CH_from_Dimension</td>
<td>Reverse of Attach_CH_to_Dimension.</td>
<td>A diachron:CodeList property instance is deleted.</td>
</tr>
<tr>
<td>Add_Instance_to_CH</td>
<td>Add a new instance into a codelist / hierarchy</td>
<td>A new diachron:CodeListTerm (actually a skos:concept) is added and attached to a codelist / hierarchy with a diachron:inScheme instance property. If there is a hierarchy, classification is defined too with a diachron:hasParent property instance.</td>
</tr>
<tr>
<td>Delete_Instance_from_CH</td>
<td>Delete an instance from a codelist / hierarchy. Reverse of Add_Instance_to_CH.</td>
<td>The structure defined in Add_Instance_to_CH is deleted.</td>
</tr>
<tr>
<td>Split_Instance_CH</td>
<td>Split an instance into n instances. If there is a hierarchy, they are at the same level.</td>
<td>A diachron:CodeListTerm instance is deleted and n diachron:CodeListTerm instances are added, assigned with a diachron:inScheme instance property. If there is a hierarchy, the added instances have the same parent instance with the one deleted (diachron:hasParent).</td>
</tr>
<tr>
<td>Merge_Instances_CH</td>
<td>n instances are merged into one. Reverse of Split_Instance_CH.</td>
<td>A diachron:CodeListTerm instance is added and n diachron:CodeListTerm instances are deleted, assigned with a diachron:inScheme instance property. If there is a hierarchy, the added instance has the same parent instance with the deleted (diachron:hasParent).</td>
</tr>
<tr>
<td>Reclassify_Instance_H</td>
<td>Change the parent instance.</td>
<td>The respective triple with predicate diachron:hasParent is deleted and another with a new diachron:CodeListTerm object is added.</td>
</tr>
<tr>
<td>Update_Instance_CH</td>
<td>Change an instance’s name.</td>
<td>The respective triple (with the suitable diachron:CodeListTerm instance as subject) is deleted and another with a new name as object is added.</td>
</tr>
</tbody>
</table>

Table 13: Proposed Simple Changes for the Multidimensional Model (Entity: Codelist/Hierarchy)
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Mapping to DIACHRON model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_Attribute</td>
<td>Add new attribute.</td>
<td>Add a new instance of diachron:AttributeProperty, with a given data type, and value.</td>
</tr>
<tr>
<td>Delete_Attribute</td>
<td>Reverse of Add_Attribute.</td>
<td>The structure described in Add_Attribute is deleted.</td>
</tr>
<tr>
<td>Modify_Attribute</td>
<td>Change the datatype of an attribute.</td>
<td>The respective triple with predicate diachron:hasDataType is deleted and another with a new datatype object is added.</td>
</tr>
<tr>
<td>Rename_Attribute</td>
<td>Rename an attribute.</td>
<td>The neighborhood of the diachron:AttributeProperty remains immutable. The mapping between uris is identified (add / delete the triple).</td>
</tr>
<tr>
<td>Update_Attribute</td>
<td>Update attribute value.</td>
<td>The respective triple is deleted and another with a new value in object is added.</td>
</tr>
</tbody>
</table>

Table 14: Proposed Simple Changes for the Multidimensional Model (Entity: Attribute)
<table>
<thead>
<tr>
<th>Change</th>
<th>Description</th>
<th>Mapping to DIACHRON model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add_Observation</td>
<td>Add an observation entity.</td>
<td>Add a new observation resource. An instance of diachron:Record is added, and is connected with diachron:RecordAttribute instances through diachron:hasRecordAttribute. Each diachron:RecordAttribute instance has a diachron:_predicate and a diachron:object property. diachron:Record is connected with a diachron:subject property with the resource.</td>
</tr>
<tr>
<td>Delete_Observation</td>
<td>Delete an observation entity. Reverse of Add_Observation.</td>
<td>The structure described in Add_Observation is deleted.</td>
</tr>
<tr>
<td>Update_Observation</td>
<td>A measure value is updated.</td>
<td>The respective triple with predicate diachron:object is deleted and another with a new value in object is added.</td>
</tr>
<tr>
<td>Add_Measure_Value_to_Observation</td>
<td>Add value to a measure in a specific observation. This is valid after Attach_Measure_to_FT.</td>
<td>Add a new diachron:RecordAttribute instance, with diachron: predicate and diachron:object, related with the respective diachron:Record instance through a diachron:hasRecordAttribute property.</td>
</tr>
<tr>
<td>Delete_Measure_Value_from_Observation</td>
<td>Reverse of Add_Measure_Value_to_Observation</td>
<td>The structure described in Add_Measure_Value_to_Observation is deleted.</td>
</tr>
<tr>
<td>Add_Dimension_Value_to_Observation</td>
<td>Add value to a dimension in a specific observation. This is valid after Attach_Dimension_to_FT.</td>
<td>Similar to Add_Measure_Value_to_Observation.</td>
</tr>
<tr>
<td>Delete_Dimension_Value_from_Observation</td>
<td>Reverse of Add_Dimension_Value_to_Observation</td>
<td>Similar to Delete_Measure_Value_from_Observation.</td>
</tr>
</tbody>
</table>

Table 15: Proposed Simple Changes for the Multidimensional Model (Entity: Observation)
B.3 RDF(S) Model

In the RDF(S) model (scientific pilot), the situation is different compared to the other pilots, because we already have a small list of confirmed and fully defined simple changes. These are presented in Subsection B.3.1 below. In addition to that, thanks to the fact that the scientific pilot employs the RDF(S) model, we can (at least partly) reuse the list of changes appearing in [34], which are generic RDF(S) changes. These changes are formally defined, and correspond to the composite and heuristic changes in the terminology of [34]. This list of changes appears in Subsection B.3.2.

B.3.1 Simple Changes Proposed by the Scientific Pilot

In this subsection we present a list of changes that have been agreed upon with the scientific pilot. The definition of these changes follows the methodology that we presented in Subsection 2.1.4, so it is broken down in three tables, which are essentially extensions of Tables 2, 3 and 4 that were presented in Subsection 2.1.4.

<table>
<thead>
<tr>
<th>Change with parameters</th>
<th>Priority</th>
<th>Parameters (in details)</th>
<th>Intuition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit_label(A, “X”, “Y”)</td>
<td>2</td>
<td>A: the class which obtained the label X: the previous label Y: the new label</td>
<td>This corresponds to the deletion of a triple that associates A with a label X and to the addition of a new triple that associates A with a label Y</td>
</tr>
<tr>
<td>Mark_as_obsolete(A)</td>
<td>1</td>
<td>A: the class which is rendered obsolete</td>
<td>This corresponds to the subsumption of A with the class Obsolete</td>
</tr>
<tr>
<td>Add_new_label(A, “label”)</td>
<td>3</td>
<td>A: the class which obtained the label label: the new label</td>
<td>This corresponds to the addition of a triple associating A with a new label</td>
</tr>
<tr>
<td>Delete_label(A, “label”)</td>
<td>3</td>
<td>A: the class which obtained the label label: the deleted label</td>
<td>This corresponds to the deletion of a triple associating A with a label</td>
</tr>
</tbody>
</table>

Table 16: High Level Information on the Simple Changes of the Scientific Pilot

<table>
<thead>
<tr>
<th>Change with parameters</th>
<th>Effects in pilots model</th>
<th>Presence</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mark_as_obsolete(A)</td>
<td>added: (A, rdfs : subClassOf, Obsolete)</td>
<td>mandatory</td>
<td>N/A</td>
</tr>
<tr>
<td>Add_new_label(A, “label”)</td>
<td>added: (A, rdfs : label, “label”)</td>
<td>mandatory</td>
<td>N/A</td>
</tr>
<tr>
<td>Delete_label(A, “label”)</td>
<td>deleted: (A, rdfs : label, “label”)</td>
<td>mandatory</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 17: Simple Changes in the Model of the Scientific Pilot
<table>
<thead>
<tr>
<th>Change with parameters</th>
<th>Effects in DIACHRON's model</th>
<th>Presence</th>
<th>Condition</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Edit_label(A, “X”, “Y”)</td>
<td>added: (R1, diachron : subject, A) (R1, diachron : hasAttribute, RAtt1) (RAtt1, diachron : predicate, rdfs : label) (RAtt1, diachron : object, “Y”)</td>
<td>mandatory</td>
<td>mandatory</td>
<td>mandatory</td>
</tr>
<tr>
<td></td>
<td>deleted: (R1, diachron : subject, A) (R1, diachron : hasAttribute, RAtt1) (RAtt1, diachron : predicate, rdfs : label) (RAtt1, diachron : object, “X”)</td>
<td>mandatory</td>
<td>mandatory</td>
<td>mandatory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mark_as_obsolete(A)</td>
<td>added: (R1, diachron : subject, A) (R1, diachron : hasAttribute, RAtt1) (RAtt1, diachron : predicate, rdfs : subClassOf) (RAtt1, diachron : object, Obsolete)</td>
<td></td>
<td></td>
<td>mandatory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Add_new_label(A, “label”)</td>
<td>added: (R1, diachron : subject, A) (R1, diachron : hasAttribute, RAtt1) (RAtt1, diachron : predicate, rdfs : label) (RAtt1, diachron : object, “label”)</td>
<td>mandatory</td>
<td>mandatory</td>
<td>mandatory</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Delete_label(A, “label”)</td>
<td>deleted: (R1, diachron : subject, A) (R1, diachron : hasAttribute, RAtt1) (RAtt1, diachron : predicate, rdfs : label) (RAtt1, diachron : object, “label”)</td>
<td>mandatory</td>
<td>mandatory</td>
<td>mandatory</td>
</tr>
</tbody>
</table>

Table 18: The Scientific Pilot’s Simple Changes in DIACHRON’s Model
### Other Proposed Simple Changes

In this subsection we present another list of proposed changes that is taken from [34]. As already mentioned, the changes presented here are generic composite and heuristic changes (in the terminology of [34]), that can be used in any RDF(S) dataset, including the datasets of the scientific pilot. For this reason, we propose this list to play the role of the simple changes for the scientific pilot (in addition to those proposed in Subsection B.3.1 above). Unlike the corresponding lists in other pilots, this list is formally defined. The changes in the list are, as usual, organized according to the entities of the RDF(S) data model, namely: *class*, *property* and *individual*.

In the following tables, for each proposed simple change, we describe:

- Its name.
- The intuition it captures, described in terms of the RDF(S) model.
- Its parameters, and the intuition behind each parameter.
- The low-level changes \((\delta^+, \delta^-)\) which are required to be present in the low-level delta \((\Delta^+, \Delta^-)\) respectively in order for said change to be detected.
- A mapping that must be present in some of the cases between URIs in the old and new versions \((\mathcal{M})\).
- A logical condition \((\phi)\) which must be satisfied in order for the change to be detected. For simplicity, the conditions in the following tables use clauses like “\(a\) is a class in \(V_1\)”., rather than the more verbose (and formal) statement: \((a, \text{rdf:type}, \text{rdfs:class}) \in V_1 \land (a, \text{rdfs:subClassOf}, \text{rdfs:resource}) \in V_1\).
Class Entity:

<table>
<thead>
<tr>
<th>Change</th>
<th>( \text{Add Class}(P_1, P_2, P_3, P_4, P_5, P_6) )</th>
<th>( \text{Delete Class}(P_1, P_2, P_3, P_4, P_5, P_6) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Add class ( a ) with its neighbourhood links</td>
<td>Delete class ( a ) with its neighbourhood links</td>
</tr>
<tr>
<td>Parameters</td>
<td>( P_1 = ) set of new parent classes of ( a ), ( P_2 = ) set of classes that have as parent ( a ), ( P_3 = ) set of new metaclasses of ( a ), ( P_4 = ) set of new individuals that are type of ( a ), ( P_5 = ) set of new labels of ( a ), ( P_6 = ) set of new comments of ( a )</td>
<td>( P_1 = ) set of old parent classes of ( a ), ( P_2 = ) set of classes that had as parent ( a ), ( P_3 = ) set of old metaclasses of ( a ), ( P_4 = ) set of individuals that were type of ( a ), ( P_5 = ) set of old comments of ( a ), ( P_6 = ) set of old labels of ( a )</td>
</tr>
<tr>
<td>( \delta^+ )</td>
<td>( \forall p \in P_1 : (a, \text{rdfs} : \text{subClassOf}, p) ), ( \forall p \in P_2 : (p, \text{rdfs} : \text{subClassOf}, a) ), ( \forall p \in P_3 : (a, \text{rdf} : \text{type}, p) ), ( \forall p \in P_4 : (p, \text{rdf} : \text{type}, a) ), ( \forall p \in P_5 : (a, \text{rdfs} : \text{comment}, p) ), ( \forall p \in P_6 : (a, \text{rdfs} : \text{label}, p) ), ( (a, \text{rdf} : \text{type}, \text{rdfs} : \text{class}) ), ( (a, \text{rdfs} : \text{subClassOf}, \text{rdfs} : \text{resource}) )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \delta^- )</td>
<td>( \emptyset )</td>
<td>( \forall p \in P_1 : (a, \text{rdfs} : \text{subClassOf}, p) ), ( \forall p \in P_2 : (p, \text{rdfs} : \text{subClassOf}, a) ), ( \forall p \in P_3 : (a, \text{rdf} : \text{type}, p) ), ( \forall p \in P_4 : (p, \text{rdf} : \text{type}, a) ), ( \forall p \in P_5 : (a, \text{rdfs} : \text{comment}, p) ), ( \forall p \in P_6 : (a, \text{rdfs} : \text{label}, p) ), ( (a, \text{rdf} : \text{type}, \text{rdfs} : \text{class}) ), ( (a, \text{rdfs} : \text{subClassOf}, \text{rdfs} : \text{resource}) )</td>
</tr>
<tr>
<td>( \mathcal{M} )</td>
<td>( \emptyset )</td>
<td>( \emptyset )</td>
</tr>
<tr>
<td>( \phi )</td>
<td>( a ) does not appear in ( V_1 ), ( \forall p \in P_1 \cup P_2 : p ) is a schema class in ( V_1 ), ( \forall p \in P_2 : p ) is a metaclass in ( V_1 ), ( \forall p \in P_2 : p ) is an individual in ( V_1 )</td>
<td>( a ) does not appear in ( V_2 ), ( \forall p \in P_1 \cup P_2 : p ) is a schema class in ( V_2 ), ( \forall p \in P_2 : p ) is a metaclass in ( V_2 ), ( \forall p \in P_2 : p ) is an individual in ( V_2 )</td>
</tr>
</tbody>
</table>
### Change

<table>
<thead>
<tr>
<th><strong>Pull_up_Class</strong>(a,B₁,B₂)</th>
<th><strong>Pull_down_Class</strong>(a,B₁,B₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Move class a to a higher position in the subsumption hierarchy</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>B₁ = set of old parents of a, B₂ = set of new parents of a</td>
</tr>
<tr>
<td>δ⁺</td>
<td>∀b₂ ∈ B₂ : (a, rdfs : subClassOf, b₂)</td>
</tr>
<tr>
<td>δ⁻</td>
<td>∀b₁ ∈ B₁ : (a, rdfs : subClassOf, b₁)</td>
</tr>
</tbody>
</table>

### Move_Class(a,B₁,B₂)

<table>
<thead>
<tr>
<th><strong>Intuition</strong></th>
<th>Move a class to a different subsumption hierarchy</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameters</strong></td>
<td>B₁ = set of old parents of a, B₂ = set of new parents of a</td>
</tr>
<tr>
<td>δ⁺</td>
<td>∀b₂ ∈ B₂ : (a, rdfs : subClassOf, b₂)</td>
</tr>
<tr>
<td>δ⁻</td>
<td>∀b₁ ∈ B₁ : (a, rdfs : subClassOf, b₁)</td>
</tr>
</tbody>
</table>

### Change_Superclasses(a,B₁,B₂)

<table>
<thead>
<tr>
<th><strong>Intuition</strong></th>
<th>Change the parents of class a</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameters</strong></td>
<td>B₁ = set of old parents of a, B₂ = set of new parents of a</td>
</tr>
<tr>
<td>δ⁺</td>
<td>∀b₂ ∈ B₂ : (a, rdfs : subClassOf, b₂)</td>
</tr>
<tr>
<td>δ⁻</td>
<td>∀b₁ ∈ B₁ : (a, rdfs : subClassOf, b₁)</td>
</tr>
</tbody>
</table>

### Group_Classes(A,b)

<table>
<thead>
<tr>
<th><strong>Intuition</strong></th>
<th>Group classes in A under b</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameters</strong></td>
<td>A = set of classes that have as new parent b, b = new parent class</td>
</tr>
<tr>
<td>δ⁺</td>
<td>∀a ∈ A : (a, rdfs : subClassOf, b)</td>
</tr>
<tr>
<td>δ⁻</td>
<td>θ</td>
</tr>
</tbody>
</table>

### Ungroup_Classes(A,b)

<table>
<thead>
<tr>
<th><strong>Intuition</strong></th>
<th>Ungroup classes in A</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Parameters</strong></td>
<td>A = set of classes that had as parent b, b = old parent class</td>
</tr>
<tr>
<td>δ⁺</td>
<td>θ</td>
</tr>
<tr>
<td>δ⁻</td>
<td>∀a ∈ A : (a, rdfs : subClassOf, b)</td>
</tr>
</tbody>
</table>
### Change

**Rename Class**\((a, b)\)

<table>
<thead>
<tr>
<th>Intuition</th>
<th>Rename class (a) to (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>(a = ) the old name of the class, (b = ) the new name of the class</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>((b, \text{rdf:} \text{type}, \text{rdfs:} \text{class}), (b, \text{rdfs:} \text{subClassOf}, \text{rdfs:} \text{resource}))</td>
</tr>
<tr>
<td>(\delta^-)</td>
<td>((a, \text{rdf:} \text{type}, \text{rdfs:} \text{class}), (a, \text{rdfs:} \text{subClassOf}, \text{rdfs:} \text{resource}))</td>
</tr>
<tr>
<td>(\phi)</td>
<td>(a) does not appear in (V_2) (\land) (b) does not appear in (V_1)</td>
</tr>
</tbody>
</table>

### Change

**Merge Classes**\((A, b)\)

<table>
<thead>
<tr>
<th>Intuition</th>
<th>Merge classes contained in (A) into (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>(A = ) the set of old names of the classes, (b = ) the new name of the class</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>((b, \text{rdf:} \text{type}, \text{rdfs:} \text{class}), (b, \text{rdfs:} \text{subClassOf}, \text{rdfs:} \text{resource}))</td>
</tr>
<tr>
<td>(\delta^-)</td>
<td>(\forall a \in A \setminus {b} : (a, \text{rdf:} \text{type}, \text{rdfs:} \text{class}), (a, \text{rdfs:} \text{subClassOf}, \text{rdfs:} \text{resource}))</td>
</tr>
<tr>
<td>(\phi)</td>
<td>(\forall a \in A \setminus {b} ) (a) does not appear in (V_2) (\land) (b) does not appear in (V_1)</td>
</tr>
</tbody>
</table>

### Change

**Split Class**\((a, B)\)

<table>
<thead>
<tr>
<th>Intuition</th>
<th>Split class (a) into classes contained in (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>(a = ) the old name of the class, (B = ) the set of new names of the classes</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>((b, \text{rdf:} \text{type}, \text{rdfs:} \text{class}), (b, \text{rdfs:} \text{subClassOf}, \text{rdfs:} \text{resource}))</td>
</tr>
<tr>
<td>(\delta^-)</td>
<td>(\forall b \in B \setminus {a} : (b, \text{rdf:} \text{type}, \text{rdfs:} \text{class}), (b, \text{rdfs:} \text{subClassOf}, \text{rdfs:} \text{resource}))</td>
</tr>
<tr>
<td>(\phi)</td>
<td>(\forall b \in B \setminus {a} ) (b) does not appear in (V_2) (\land) (\forall b \in B ) (b) does not appear in (V_1)</td>
</tr>
</tbody>
</table>

### Change

**Merge Classes Into Existing**\((A, b)\)

<table>
<thead>
<tr>
<th>Intuition</th>
<th>Merge classes contained in (A) into (b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>(A = ) the set of old names of the classes, (b = ) the new name of the class</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>(\forall a \in A \setminus {b} : (a, \text{rdf:} \text{type}, \text{rdfs:} \text{class}), (a, \text{rdfs:} \text{subClassOf}, \text{rdfs:} \text{resource}))</td>
</tr>
<tr>
<td>(\delta^-)</td>
<td>(\forall a \in A \setminus {b}, \forall b \in B,</td>
</tr>
<tr>
<td>(\phi)</td>
<td>(\forall a \in A \setminus {b} ) (a) does not appear in (V_2) (\land) (b) is a schema class in both (V_1) and (V_2)</td>
</tr>
</tbody>
</table>

### Change

**Split Class Into Existing**\((a, B)\)

<table>
<thead>
<tr>
<th>Intuition</th>
<th>Split class (a) into classes contained in (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameters</td>
<td>(a = ) the old name of the class, (B = ) the set of new names of the classes</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>(\forall b \in B \setminus {a} : (b, \text{rdf:} \text{type}, \text{rdfs:} \text{class}), (b, \text{rdfs:} \text{subClassOf}, \text{rdfs:} \text{resource}))</td>
</tr>
<tr>
<td>(\delta^-)</td>
<td>(\forall a \in A \setminus {b}, \forall b \in B,</td>
</tr>
<tr>
<td>(\phi)</td>
<td>(\forall a \in A \setminus {b} ) (a) is a schema class in both (V_1) and (V_2) (\land) (\forall b \in B \setminus {a} ) (b) does not appear in (V_1)</td>
</tr>
</tbody>
</table>
### Property Entity:

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Property</th>
<th>Delete Property</th>
</tr>
</thead>
<tbody>
<tr>
<td><em>(a, P₁, P₂, P₃, P₄, P₅, P₆, P₇, P₈, P₉, P₁₀)</em></td>
<td>Add property a with its neighbourhood links</td>
<td>Delete property a with its neighbourhood links</td>
</tr>
</tbody>
</table>

#### Parameters

- $P₁$ = set of new parent properties of a,
- $P₂$ = set of new properties that have as parent a,
- $P₃$ = set of new metaproperties of a,
- $P₄$ = set of pairs of new property instances of a,
- $P₅$ = the new domain of a,
- $P₆$ = the new range of a,
- $P₇$ = set of new comments of a,
- $P₈$ = set of new labels of a.

#### $δ⁺$

- $∀p ∈ P₁ : (a, rdfs : subPropertyOf, p),$
- $∀p ∈ P₂ : (p, rdfs : subPropertyOf, a),$
- $∀p ∈ P₃ : (a, rdf : type, p),$
- $(p₁, p₂) ∈ P₄ : (p₁, a, p₂),$
- $(a, rdfs : domain, p₅),$
- $(a, rdfs : range, p₆),$
- $∀p ∈ P₇ : (a, rdfs : comment, p),$
- $∀p ∈ P₈ : (a, rdfs : label, p),$
- $(a, rdf : type, rdf : property)$

#### $δ⁻$

- $∀p ∈ P₁ : (a, rdfs : subPropertyOf, p),$
- $∀p ∈ P₂ : (p, rdfs : subPropertyOf, a),$
- $∀p ∈ P₃ : (a, rdf : type, p),$
- $(p₁, p₂) ∈ P₄ : (p₁, a, p₂),$
- $∀p ∈ P₅ : (a, rdfs : comment, p),$
- $∀p ∈ P₆ : (a, rdfs : label, p),$
- $(a, rdfs : domain, p₅),$
- $(a, rdfs : range, p₆),$
- $(a, rdf : type, rdf : property)$

#### $φ$ *(a does not exist in $V₁)*

- $∀p ∈ P₁ ∪ P₂ : p$ is a property in $V₁$ *
- $∀p ∈ P₃ : p$ is a metaproperty in $V₁$ *
- $(p₁, p₂) ∈ P₄ : p₁$ is of the same type in both $V₁, V₂$, and $p₂$ is of the same type in both $V₁, V₂$ *
- $P₅$ is of the same type in both $V₁, V₂$ *
- $P₆$ is of the same type in both $V₁, V₂$ *

#### $φ$ *(a does not exist in $V₂)*

- $∀p ∈ P₁ ∪ P₂ : p$ is a property in $V₂$ *
- $∀p ∈ P₃ : p$ is a metaproperty in $V₂$ *
- $(p₁, p₂) ∈ P₄ : p₁$ is of the same type in both $V₁, V₂$, and $p₂$ is of the same type in both $V₁, V₂$ *
- $P₅$ is of the same type in both $V₁, V₂$ *
- $P₆$ is of the same type in both $V₁, V₂$ *
<table>
<thead>
<tr>
<th>Change</th>
<th>Pull-up Property((a,B_1,B_2))</th>
<th>Pull-down Property((a,B_1,B_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Move property (a) to a higher position in the subsumption hierarchy</td>
<td>Move property (a) to a lower position in the subsumption hierarchy</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>(B_1) = set of old parents of (a), (B_2) = set of new parents of (a)</td>
<td>(B_1) = set of old parents of (a), (B_2) = set of new parents of (a)</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>(\forall b \in B_1 : (a, \text{rdfs: subPropertyOf}, b_2))</td>
<td>(\forall b \in B_2 : (a, \text{rdfs: subPropertyOf}, b_2))</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>(\forall b \in B_1 : (a, \text{rdfs: subPropertyOf}, b_1))</td>
<td>(\forall b \in B_1 : (a, \text{rdfs: subPropertyOf}, b_1))</td>
</tr>
<tr>
<td>(\Delta)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\phi)</td>
<td>(a) is a property in both (V_1) and (V_2) (\land) (\forall b \in B_1 \cup B_2 \ b) is a property in both (V_1) and (V_2) (\land) (\forall b_1 \in B_1, \forall b_2 \in B_2 : (b_1, \text{rdfs: subPropertyOf}, b_2) \in C(V_1) \land (b_1, \text{rdfs: subPropertyOf}, b_1) \notin C(V_2))</td>
<td>(a) is a property in both (V_1) and (V_2) (\land) (\forall b \in B_1 \cup B_2 \ b) is a property in both (V_1) and (V_2) (\land) (\forall b_1 \in B_1, \forall b_2 \in B_2 : (b_2, \text{rdfs: subPropertyOf}, b_1) \notin C(V_1) \land (b_2, \text{rdfs: subPropertyOf}, b_2) \in C(V_2))</td>
</tr>
<tr>
<td><strong>Change</strong></td>
<td>Move property (a) to a different subsumption hierarchy</td>
<td>Change the parents of property (a)</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>(B_1) = set of old parents of (a), (B_2) = set of new parents of (a)</td>
<td>(B_1) = set of old parents of (a), (B_2) = set of new parents of (a)</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>(\forall b \in B_2 : (a, \text{rdfs: subClassOf}, b_2))</td>
<td>(\forall b \in B_2 : (a, \text{rdfs: subClassOf}, b_2))</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>(\forall b \in B_1 : (a, \text{rdfs: subClassOf}, b_1))</td>
<td>(\forall b \in B_1 : (a, \text{rdfs: subClassOf}, b_1))</td>
</tr>
<tr>
<td>(\Delta)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\phi)</td>
<td>(a) is a property in both (V_1) and (V_2) (\land) (\forall b \in B_1 \cup B_2 \ b) is a property in both (V_1) and (V_2) (\land) (\forall b_1 \in B_1, \forall b_2 \in B_2 : (b_2, \text{rdfs: subPropertyOf}, b_1) \notin C(V_1) \land (b_2, \text{rdfs: subPropertyOf}, b_2) \in C(V_2))</td>
<td>(a) is a property in both (V_1) and (V_2) (\land) (\forall b \in B_1 \cup B_2 \ b) is a property in both (V_1) and (V_2) (\land) (\forall b_1 \in B_1, \forall b_2 \in B_2 : (b_2, \text{rdfs: subPropertyOf}, b_1) \notin C(V_1) \land (b_2, \text{rdfs: subPropertyOf}, b_2) \in C(V_2))</td>
</tr>
<tr>
<td><strong>Change</strong></td>
<td>Group-Properties-Under((A,b))</td>
<td>Ungroup-Properties-Under((A,b))</td>
</tr>
<tr>
<td><strong>Intuition</strong></td>
<td>Group properties in (A) under (b)</td>
<td>Ungroup properties in (A) under (b)</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>(A) = set of properties that have as new parent (b), (b) = new parent property (b)</td>
<td>(A) = set of properties that had as parent (b), (b) = the old parent property (b)</td>
</tr>
<tr>
<td>(\delta^+)</td>
<td>(\forall a \in A : (a, \text{rdfs: subPropertyOf}, b))</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\sigma)</td>
<td>(\forall a \in A : (a, \text{rdfs: subPropertyOf}, b))</td>
<td>(\forall a \in A : (a, \text{rdfs: subPropertyOf}, b))</td>
</tr>
<tr>
<td>(\Delta)</td>
<td>(\emptyset)</td>
<td>(\emptyset)</td>
</tr>
<tr>
<td>(\phi)</td>
<td>(\forall a \in A : (a, \text{rdfs: subPropertyOf}, x) \in C(V_1) \rightarrow (a, \text{rdfs: subPropertyOf}, x) \in C(V_2))</td>
<td>(\forall a \in A : (a, \text{rdfs: subPropertyOf}, x) \in C(V_1) \rightarrow (a, \text{rdfs: subPropertyOf}, x) \in C(V_2))</td>
</tr>
</tbody>
</table>

Grant Agreement No. 601043
The changes `Specialize_Range` and `Generalize_Range` are defined analogously by replacing `rdfs:domain` with `rdfs:range` in all positions.

The change `Change_Range` is defined analogously by replacing `rdfs:domain` with `rdfs:range` in all positions.

The change `Rename_Individual` is defined analogously, by replacing `rdf:property` with `rdfs:resource` in $\delta^+, \delta^-$. 

---

<table>
<thead>
<tr>
<th>Change</th>
<th><code>Specialize_Domain(a, b_1, b_2)</code></th>
<th><code>Generalize_Domain(a, b_1, b_2)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Change the domain of property $a$ to a subclass of it</td>
<td>Change the domain of property $a$ to a superclass of it</td>
</tr>
<tr>
<td>Parameters</td>
<td>$b_1 = \text{old domain of } a$, $b_2 = \text{new domain of } a$</td>
<td>$b_1 = \text{old domain of } a$, $b_2 = \text{new domain of } a$</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>$(a, \text{rdfs:domain}, b_1)$</td>
<td>$(a, \text{rdfs:domain}, b_2)$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$(a, \text{rdfs:domain}, b_1)$</td>
<td>$(a, \text{rdfs:domain}, b_2)$</td>
</tr>
</tbody>
</table>

[...]

<table>
<thead>
<tr>
<th>Change</th>
<th><code>Change_Domain(a, b_1, b_2)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Change the domain of property $a$.</td>
</tr>
<tr>
<td>Parameters</td>
<td>$b_1 = \text{old domain of } a$, $b_2 = \text{new domain of } a$</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>$(a, \text{rdfs:domain}, b_1)$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$(a, \text{rdfs:domain}, b_1)$</td>
</tr>
</tbody>
</table>

[...]

<table>
<thead>
<tr>
<th>Change</th>
<th><code>Change_Range(a, b_1, b_2)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Change the range of property $a$ to an object</td>
</tr>
<tr>
<td>Parameters</td>
<td>$b_1 = \text{old range of } a$, $b_2 = \text{new range of } a$</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>$(a, \text{rdfs:range}, b_1)$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$(a, \text{rdfs:range}, b_1)$</td>
</tr>
</tbody>
</table>

[...]

<table>
<thead>
<tr>
<th>Change</th>
<th><code>Change_Comment(u, a, b)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Change comment of resource $u$ from $a$ to $b$</td>
</tr>
<tr>
<td>Parameters</td>
<td>$a = \text{the old comment}$, $b = \text{the new comment}$</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>$(u, \text{rdfs:comment}, b)$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$(u, \text{rdfs:comment}, a)$</td>
</tr>
</tbody>
</table>

[...]

<table>
<thead>
<tr>
<th>Change</th>
<th><code>Rename_Property(a, b)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Rename property $a$ to $b$</td>
</tr>
<tr>
<td>Parameters</td>
<td>$a = \text{the old name of the property}$, $b = \text{the new name of the property}$</td>
</tr>
<tr>
<td>$\delta^+$</td>
<td>$(b, \text{rdf:type}, \text{rdfs:property})$</td>
</tr>
<tr>
<td>$\delta^-$</td>
<td>$(a, \text{rdf:type}, \text{rdfs:property})$</td>
</tr>
</tbody>
</table>

[...]

Grant Agreement No. 601043
Intuition

Merge properties contained in $A$ into $b$

Split property $a$ into properties contained in $B$

Parameters

$A$ = the set of old names of the properties,
$b$ = the new name of the property

$a$ = the old name of the property,
$B$ = the set of new names of the properties

$\delta^+$

$b : \text{rdf:property}$

$\forall b \in B : \{b, \text{rdf:property}\}$

$\delta^-$

$b \in A \Rightarrow \{a, \text{rdf:property}\}$

$a \in A \Rightarrow \text{NDR}$

$\phi$

$a \in A \Rightarrow \{b\}$

$b \in B \Rightarrow \{a\}$

$\phi$

$a \in A \Rightarrow \{b\}$

$b \in B \Rightarrow \{a\}$

The changes $\text{Merge\_Individuals\_Into\_Existing}$ and $\text{Split\_Individual\_Into\_Existing}$ are defined analogously, by replacing $\text{rdf:property}$ with $\text{rdfs:resource}$ in $\delta^+ , \delta^-$. 

The changes $\text{Merge\_Properties\_Into\_Existing}$ and $\text{Split\_Property\_Into\_Existing}$ are defined analogously, by replacing $\text{rdf:property}$ with $\text{rdfs:resource}$ in $\delta^+ , \delta^-$. 

The changes $\text{Merge\_Individuals\_Into\_Existing}$ and $\text{Split\_Individual\_Into\_Existing}$ are defined analogously, by replacing $\text{rdf:property}$ with $\text{rdfs:resource}$ in $\delta^+ , \delta^-$ and adapting the conditions to refer to the proper type.
### Individual Entity:

<table>
<thead>
<tr>
<th>Change</th>
<th>Add Individual$⁺(a,P₁,P₂,P₃)$</th>
<th>Delete Individual$⁺(a,P₁,P₂,P₃)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Add individual $a$ with its neighbourhood links</td>
<td>Delete individual $a$ with its neighbourhood links</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>$P₁ = \text{set of new classes of } a$, $P₂ = \text{set of new comments of } a$, $P₃ = \text{set of new labels of } a$</td>
<td>$P₁ = \text{set of old classes of } a$, $P₂ = \text{set of old comments of } a$, $P₃ = \text{set of old labels of } a$</td>
</tr>
</tbody>
</table>

\[ \delta^⁺ \]
\[ \forall p ∈ P₁ : (a, \text{rdf} : \text{type}, p), \]
\[ \forall p ∈ P₂ : (a, \text{rdfs} : \text{comment}, p), \]
\[ \forall p ∈ P₃ : (a, \text{rdfs} : \text{label}, p), \]
\[ (a, \text{rdf} : \text{type}, \text{rdfs} : \text{resource}) \]

\[ \delta^⁻ \]
\[ \emptyset \]

\[ \emptyset \]
\[ a \text{ does not appear in } V₁ \land \]
\[ \forall p ∈ P₁ \ p \text{ is a schema class in } V₁ \]
\[ a \text{ does not appear in } V₂ \land \]
\[ \forall p ∈ P₁ \ p \text{ is a schema class in } V₂ \]

### Reclassify Individual$⁺$ Higher

<table>
<thead>
<tr>
<th>Change</th>
<th>Reclassify Individual$⁺$ Higher $(a,B₁,B₂)$</th>
<th>Reclassify Individual$⁺$ Lower $(a,B₁,B₂)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Intuition</strong></td>
<td>Reclassify an individual under a class that is at a higher position in the subsumption hierarchy</td>
<td>Reclassify an individual under a class that is at a lower position in the subsumption hierarchy</td>
</tr>
<tr>
<td><strong>Parameters</strong></td>
<td>$B₁ = \text{set of old classes instantiating } a$, $B₂ = \text{set of new classes instantiating } a$</td>
<td>$B₁ = \text{set of old classes instantiating } a$, $B₂ = \text{set of new classes instantiating } a$</td>
</tr>
</tbody>
</table>

\[ \delta^⁺ \]
\[ \forall b₂ ∈ B₂ : (a, \text{rdf} : \text{type}, b₂) \]

\[ \delta^⁻ \]
\[ \forall b₁ ∈ B₁ : (a, \text{rdf} : \text{type}, b₁) \]

\[ \emptyset \]
\[ a \text{ is an individual in both } V₁ \text{ and } V₂ \land \]
\[ \forall b₁ ∈ B₁ \cup B₂ \ b \text{ is a schema class in both } V₁ \text{ and } V₂ \land \]
\[ \forall b₂ ∈ B₂ : (a, \text{rdf} : \text{type}, b₂) ∈ \text{Cl}(V₁) \land \]
\[ (b₁, \text{rdfs} : \text{subClassOf}, b₂) ∈ \text{Cl}(V₂) \land \]
\[ \forall b ∈ B₁ \cup B₂ : (a, \text{rdf} : \text{type}, b) ∈ V₁ \leftrightarrow (a, \text{rdf} : \text{type}, b) ∈ V₂ \land \]
\[ B₁ \neq \emptyset \land \]
\[ B₂ \neq \emptyset \]

\[ \emptyset \]
\[ a \text{ is an individual in both } V₁ \text{ and } V₂ \land \]
\[ \forall b ∈ B₁ \cup B₂ \ b \text{ is a schema class in both } V₁ \text{ and } V₂ \land \]
\[ \forall b₁ ∈ B₁ \cup B₂ : (b₂, \text{rdfs} : \text{subClassOf}, b₁) ∈ \text{Cl}(V₁) \land \]
\[ (b₂, \text{rdfs} : \text{subClassOf}, b₁) ∈ \text{Cl}(V₂) \land \]
\[ \forall b ∈ B₁ \cup B₂ : (a, \text{rdf} : \text{type}, b) ∈ V₁ \leftrightarrow (a, \text{rdf} : \text{type}, b) ∈ V₂ \land \]
\[ B₁ \neq \emptyset \land \]
\[ B₂ \neq \emptyset \]

The changes Reclassify Class$⁺$ Higher, Reclassify Class$⁺$ Lower, Reclassify Property$⁺$ Higher and Reclassify Property$⁺$ Lower are defined analogously with the exception that the typing requirements in the conditions should be adapted analogously.

---

Grant Agreement No. 601043

86
<table>
<thead>
<tr>
<th>Change</th>
<th>Reclassify Individual(a,B₁,B₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuition</td>
<td>Reclassify an individual</td>
</tr>
<tr>
<td>Parameters</td>
<td>B₁ = set of old classes instantiating a, B₂ = set of new classes instantiating a</td>
</tr>
<tr>
<td>δ⁺</td>
<td>∀b₂ ∈ B₂ : (a, rdf : type, b₂)</td>
</tr>
<tr>
<td>δ⁻</td>
<td>∀b₁ ∈ B₁ : (a, rdf : type, b₁)</td>
</tr>
<tr>
<td>ϕ</td>
<td>a is an individual in both V₁ and V₂∧</td>
</tr>
<tr>
<td></td>
<td>∀b ∈ B₁ ∪ B₂ b is a schema class in both V₁ and V₂∧</td>
</tr>
<tr>
<td></td>
<td>(¬φ(Reclassify Individual_Higher(a,B₁,B₂)) ∧</td>
</tr>
<tr>
<td></td>
<td>¬φ(Reclassify Individual_Lower(a,B₁,B₂))) ∧</td>
</tr>
<tr>
<td></td>
<td>B₁ ≠ ∅ ∧</td>
</tr>
<tr>
<td></td>
<td>B₂ ≠ ∅</td>
</tr>
</tbody>
</table>

The changes Reclassify-Class and Reclassify-Property are defined analogously with the exception that the typing requirements in the conditions should be adapted analogously.
Table 19 below shows the types of constraints that are supported by $DL$-$Lite_A$. In particular, the first column shows the supported constraints written using standard DL notation, the second column shows their corresponding logical form as a Disjunctive Embedded Dependency (DED [18]), and the third column provides the intuition behind said constraint. Note that this table contains only negative inclusions, i.e., those that can be violated under the Open World Assumption (OWA).

<table>
<thead>
<tr>
<th>Constraint Form</th>
<th>DED formulation</th>
<th>Intuitive Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A \sqsubseteq \neg B$</td>
<td>$\forall x A(x) \land B(x) \rightarrow \bot$</td>
<td>$A$ and $B$ are disjoint</td>
</tr>
<tr>
<td>$\exists P \sqsubseteq \neg B$</td>
<td>$\forall x, y P(x, y) \land B(x) \rightarrow \bot$</td>
<td>The domain of $P$ is disjoint with $B$</td>
</tr>
<tr>
<td>$\exists P^\neg \sqsubseteq \neg B$</td>
<td>$\forall x, y P(x, y) \land B(x) \rightarrow \bot$</td>
<td>The range of $P$ is disjoint with $B$</td>
</tr>
<tr>
<td>$A \sqsubseteq \neg \exists P$</td>
<td>$\forall x, y A(x) \land \exists P(x, y) \rightarrow \bot$</td>
<td>$A$ is disjoint with the domain of $P$</td>
</tr>
<tr>
<td>$A \sqsubseteq \neg \exists P^\neg$</td>
<td>$\forall x, y A(x) \land \exists P^\neg(y, x) \rightarrow \bot$</td>
<td>$A$ is disjoint with the range of $P$</td>
</tr>
<tr>
<td>$\exists R \sqsubseteq \neg \exists P$</td>
<td>$\forall x, y, z R(x, y) \land P(x, z) \rightarrow \bot$</td>
<td>The domain of $R$ is disjoint with the domain of $P$</td>
</tr>
<tr>
<td>$\exists R^\neg \sqsubseteq \neg \exists P$</td>
<td>$\forall x, y, z R(y, x) \land P(z, x) \rightarrow \bot$</td>
<td>The range of $R$ is disjoint with the domain of $P$</td>
</tr>
<tr>
<td>$\exists R \sqsubseteq \neg \exists P^\neg$</td>
<td>$\forall x, y, z R(y, x) \land \exists P^\neg(z, x) \rightarrow \bot$</td>
<td>The range of $R$ is disjoint with the range of $P$</td>
</tr>
<tr>
<td>$P \sqsubseteq \neg R$</td>
<td>$\forall x, y P(x, y) \land \exists R(x, y) \rightarrow \bot$</td>
<td>$P$ and $R$ are disjoint</td>
</tr>
<tr>
<td>(funct $P$)</td>
<td>$\forall x, y, z P(x, y) \land P(x, z) \land (y \neq z) \rightarrow \bot$</td>
<td>$P$ is a functional property</td>
</tr>
<tr>
<td>(funct $P^\neg$)</td>
<td>$\forall x, y, z P(y, x) \land P(z, x) \land (y \neq z) \rightarrow \bot$</td>
<td>$P$ is an inverse functional property</td>
</tr>
</tbody>
</table>

Table 19: Constraints Supported by $DL$-$Lite_A$
## D Data Constraints by the Pilots

Table 20 contains a list of constraints that was provided (in natural language form) by the pilots in deliverable D1.1 [36]. These constraints are applied on the Experimental Factor Ontology (EFO) of the scientific pilot. In this appendix, we revisit those constraints in order to identify the service that will handle each of them, and to explain how they are formalized. In particular, Table 20 lists the natural-language description of each constraint (as it appears in [36]), the relevant service (repairing or cleaning) and some notes related to how said constraint will be handled.

<table>
<thead>
<tr>
<th>Constraint (Natural Language)</th>
<th>Relevant Service</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>The EFO input file is in valid OWL format.</td>
<td>Cleaning</td>
<td>Standard parsers can be used for that.</td>
</tr>
<tr>
<td>A single ontology version number is specified, in an <code>&lt;owl:versionInfo&gt;</code></td>
<td>Repair</td>
<td>Functional property constraint. Corresponding DL-LiteA axiom: <code>(funct owl:versionInfo)</code></td>
</tr>
<tr>
<td>Date is specified in an <code>&lt;rdfs:comment&gt;</code> element with a value prefixed by Date</td>
<td>Cleaning</td>
<td>Constraint related to the form of literals.</td>
</tr>
<tr>
<td>Every class ID is a URI formed from a valid namespace name. Valid namespaces include the default (EFO) namespace and the namespaces of acceptable imports: <a href="http://www.ebi.ac.uk/efo/">http://www.ebi.ac.uk/efo/</a> <a href="http://purl.org/obo/owl/CL#">http://purl.org/obo/owl/CL#</a> <a href="http://purl.org/obo/owl/PATO#">http://purl.org/obo/owl/PATO#</a> and so on...</td>
<td>Cleaning</td>
<td>Constraint related to the form of URIs.</td>
</tr>
<tr>
<td>EFO IDs used in class IRIs are valid. IRI fragment of IRIs in EFO namespace must have the form:EFO:XXXXXXX</td>
<td>Cleaning</td>
<td>Constraint related to the form of URIs.</td>
</tr>
<tr>
<td>Annotation of EFO URIs are valid. Where a class was imported from an ontology and replaced an existing EFO class, the new class ID will use the imported namespace and the old ID (an EFO URI) may be retained as an annotation.</td>
<td>Cleaning</td>
<td>Constraint related to the form of URIs. Access to the time dimension is required.</td>
</tr>
<tr>
<td>Every class has a single label, specified in the <code>&lt;rdfs:label&gt;</code></td>
<td>Repair</td>
<td>Functional property constraint. Corresponding DL-LiteA axiom: <code>owl:Class ⊑ ∃ rdfs:label (funct rdfs:label)</code></td>
</tr>
<tr>
<td>Class labels are unique</td>
<td>Repair</td>
<td>Inverse functional property constraint. Corresponding DL-LiteA axiom: <code>(funct rdfs:label)¬1</code></td>
</tr>
<tr>
<td>Class synonyms are unique. Synonyms are defined in the <code>&lt;alternative_term&gt;</code></td>
<td>Repair</td>
<td>Inverse functional property constraint. Corresponding DL-LiteA axiom: <code>(funct alternative_term)¬1</code></td>
</tr>
<tr>
<td>Class labels and synonyms collectively are unique. A synonym for a class should never be the label of another class.</td>
<td>Repair</td>
<td>Range disjointness constraint. Corresponding DL-LiteA axiom: <code>p(alternative_term) ⊑ ¬p(rdfs:label)</code></td>
</tr>
<tr>
<td>Constraint (Natural Language)</td>
<td>Relevant Service</td>
<td>Notes</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>-----------------</td>
<td>-------</td>
</tr>
<tr>
<td>There are not an excessive number (currently 20) of class synonyms.</td>
<td>Repair</td>
<td>Cardinality constraint. Corresponding $DL-Lite_A^N$ axiom (note that this is not a $DL-Lite_A$ axiom; its inclusion would cause an exponential explosion in the complexity of diagnosis [7]): $\leq 21 \text{alternative_term}^*\text{owl:Class}$</td>
</tr>
<tr>
<td>Obsolete classes are marked as organizational using the <code>&lt;organizational_class&gt;</code> element</td>
<td>Repair</td>
<td>Positive Inclusion. It does not cause violations because of the Open World Assumption of our framework.</td>
</tr>
<tr>
<td>Annotation (on the ontology or a class) has no value e.g. <code>&lt;alternative_term&gt; &lt;\alternative_term&gt;</code></td>
<td>Cleaning</td>
<td>Constraint related to the form of a literal; the value of the annotation cannot be the empty string.</td>
</tr>
<tr>
<td>Annotation (on the ontology or a class) has no leading or trailing whitespace</td>
<td>Cleaning</td>
<td>Constraint related to the form of a literal.</td>
</tr>
<tr>
<td>Term capitalisation. Labels or synonyms with capitals are now reported (as non-critical errors).</td>
<td>Cleaning</td>
<td>Constraint related to the form of a literal.</td>
</tr>
</tbody>
</table>

Table 20: Constraints for the Scientific Pilot

Table 21 contains a list of constraints that appear in the current version of the Experimental Factor Ontology (EFO), which was provided by the pilots. These constraints are applied on the datasets that are expressed in terms of this ontology. Table 21 describes the constraints in OWL syntax (as they appear in EFO), and in natural language, as well as their corresponding $DL-Lite_A$ axiom. The namespaces used in the table are the following:

- `PREFIX obo2: http://purl.obolibrary.org/obo#`
- `PREFIX obo: http://purl.obolibrary.org/obo/`
- `PREFIX OBO_REL: http://purl.org/obo/owl/OBO_REL#`
- `PREFIX ro: http://www.obofoundry.org/ro/ro.owl#`
<table>
<thead>
<tr>
<th>Constraint (OWL Syntax)</th>
<th>Constraint (Natural Language)</th>
<th>Corresponding $DL$-Lite$_A$ Axiom</th>
</tr>
</thead>
<tbody>
<tr>
<td>obo2:has_role rdf:type owl:InverseFunctionalProperty</td>
<td>has_role is an inverse functional property.</td>
<td>(funct obo2:has_role)</td>
</tr>
<tr>
<td>obo:OBI_0000298 rdf:type owl:InverseFunctionalProperty</td>
<td>OBI_0000298 is an inverse functional property.</td>
<td>(funct obo:OBI_0000298)</td>
</tr>
<tr>
<td>OBO_REL:bearer_of rdf:type owl:InverseFunctionalProperty</td>
<td>bearer_of is an inverse functional property.</td>
<td>(funct OBO_REL:bearer_of)</td>
</tr>
<tr>
<td>OBO_REL:inheres_in rdf:type owl:FunctionalProperty</td>
<td>inheres_in is a functional property.</td>
<td>(funct OBO_REL:inheres_in)</td>
</tr>
<tr>
<td>OBO_REL:role_of rdf:type owl:FunctionalProperty</td>
<td>role_of is a functional property.</td>
<td>(funct OBO_REL:role_of)</td>
</tr>
<tr>
<td>ro:contained_in rdf:type owl:InverseFunctionalProperty</td>
<td>contained_in is an inverse functional property.</td>
<td>(funct ro:contained_in)</td>
</tr>
<tr>
<td>ro:contains rdf:type owl:FunctionalProperty</td>
<td>contains is a functional property.</td>
<td>(funct ro:contains)</td>
</tr>
</tbody>
</table>

Table 21: Constraints Appearing in the Current Version of EFO